ABSTRACT

ZHI, RUI. Design and Evaluation of Instructional Supports for Novice Programming Environments. (Under the direction of Dr. Tiffany Barnes and Dr. Thomas Price).

Students often find programming difficult to learn. Traditionally, computer science courses expect students to learn programming by writing code to solve practice programming problems. However, prior studies in many educational domains show that novices can learn more effectively with alternative instructional supports, such as worked examples, erroneous worked examples, or self-explanation prompts. In intelligent tutoring systems and gamified educational environments, these instructional supports have been shown to improve student learning efficiency. However, less work has explored how and when these supports are effective in the domain of programming, and how they compare to problem solving (writing code from scratch). Therefore, this work investigates how different types of instructional support impact novices’ outcomes when learning to program in block-based programming environments. I also devised a framework to explain the observed differences in student outcomes when using different instructional supports, based on cognitive science and educational psychology. I focus on block-based programming environments because they are commonly used in introductory programming courses, but little work has been done to understand how the effects of instructional supports can be generalized to those environments. This work contributes to the design and incorporation of effective instructional supports into novice programming environments, which has the potential to impact many novice learners.

My primary research question is: How can we design effective instructional supports, using existing theories from cognitive science and educational psychology, to help students learn more efficiently than writing program code? In this work, I designed, evaluated and compared different instructional supports in novice programming environments. Based on the promising results of worked examples in other domains, I designed and investigated worked examples in two novice programming environments. In my first study, I evaluated instructional text, worked examples, and erroneous worked examples in a programming game with middle school students, and the results suggested that students may not actively engage in learning from worked examples and erroneous worked examples may be a promising strategy to help novices learn programming. In my second study, I evaluated incomplete worked examples with self-explanation prompts in a block-based programming environment and found that worked examples may improve students’ learning efficiency. However, the two studies suggest that there are large challenges to implement effective worked examples in the domain of programming, and that students may not actively engage in learning from worked examples. To address this issue, I designed two other instructional supports including Parsons problems and example-based feedback, which both strike a balance between worked examples and problem solving while increasing the activity engagement level. In my third
study, I present a design of Parsons problems in a novice programming environment and a study where I evaluated the Parsons problems in a classroom setting. The results show Parsons problems saved students 15 to 17 minutes (43.6% to 65.2% of total time) on 30-minute lab assignments and they performed just as well on subsequent assignments compared to writing code. Furthermore, the results show that the effectiveness of Parsons problems maybe because Parsons problems dramatically reduce the programming solution space, let students focus on solving the problem rather than having to solve the combined problem of devising a solution, searching for needed components, and composing them together.

To promote our goal of the automatic creation of adaptive instructional supports for novices solving open-ended problems, in my fourth study, I developed an algorithm to automatically generate data-driven, adaptive example-based feedback, which helps a student complete one meaningful step of a solution. Based on the evaluation from experts, the algorithm can generate more relevant examples than examples derived naively from students data, and the example quality may be improved further by leveraging interactivity with students. The results suggest that the data-driven approach could be used to generate adaptive, example-based feedback to help struggling students by leveraging student data.

In summary, my work provides promising results that effective instructional supports can be designed to help students learning programming more efficiently than writing code from scratch. I situate these results in a new framework that explains differences in learning efficiency across supports based on the ICAP activity engagement framework and Cognitive Load Theory. Specifically, my results suggest that there are large challenges to implementing effective worked examples in the domain of programming, and that students may benefit from erroneous worked examples, incomplete worked examples with self-explanation prompts, and Parsons problems. As evaluated by experts, students may also benefit from the adaptive, data-driven example-based feedback. The instructional support design in programming should consider student engagement with the activity by designing instructional supports that strike a balance between the lower cognitive load of worked examples and the higher activity engagement of problem solving.

The novel contributions of this work include: (1) design and implementation of different instructional supports in novice programming environments; (2) comparisons of the impact of instructional supports and problem solving on student learning efficiency; (3) a method for generating data-driven example-based feedback; (4) the evaluation of the instructional supports, including example-based feedback, in a novice programming environment, and (5) a new framework that explains differences in instructional supports based on cognitive load and activity engagement. The results of this work can guide researchers and instructors to develop effective instructional supports in novice programming environments, suggest ways that prior data can be used to generate these instructional supports, and provides insights on how and when the instructional supports are likely to be effective.
Design and Evaluation of Instructional Supports
for Novice Programming Environments

by
Rui Zhi

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2019

APPROVED BY:

__________________________________________  ________________________________
Dr. Min Chi                                  Dr. Eric Wiebe

__________________________________________  ________________________________
Dr. Tiffany Barnes                            Dr. Thomas Price
Co-chair of Advisory Committee                Co-chair of Advisory Committee
DEDICATION

To my grandfather, parents, sister, and brother.
Rui Zhi was born in Liupanshui, known as “The Cool City” in China. Rui started playing video games since five years old. His curiosity about video games encourages him to study Computer Science a decade later.

Rui studied Computer Science since 2010 and received his Bachelor of Engineering degree in 2014 from Beijing University of Chemical Technology. In the same year, he started his Ph.D. journey in Computer Science at North Carolina State University. He joined Dr. Tiffany Barnes’ lab in 2014 and Dr. Thomas Price’s lab in 2018. During his Ph.D., he had worked as a Data Scientist intern at Hi Fidelity Genetics (2016 Summer), Software Engineering intern at Google (2017 Summer), and Software Engineering intern at Facebook (2018 Summer). In his spare time, he likes cooking, exercising, and reading. He is also an amateur drummer.
ACKNOWLEDGEMENTS

I cannot finish this work without the help and support from other talented people, especially my advisors, Dr. Tiffany Barnes and Dr. Thomas Price. Thank you for guiding me on research and encouraging me to overcome obstacles in life.

Dr. Tiffany Barnes is an excellent advisor and role model. Dr. Barnes not only has a knowledgeable mind but also a caring heart for us. She is generous with words with kindness and encourages me a lot when I faced difficulties in research and life. The most valuable lessons I learned from her are her communication skills and attitudes towards students and working. She can describe complicated ideas in a way that a novice can understand. Dr. Barnes also organizes social activities frequently to bring and connect lab members together. I am very thankful for Dr. Barnes’ research guidance, emotional support, and encouragement.

Dr. Price is not only an advisor but also a friend to me. He is a brilliant and responsible mentor. The way he does research serves as a “worked example” and I learned a lot from him. Dr. Price also provides me with concrete suggestions on improving my work. I really appreciate the opportunity to study and work with him. I am grateful to Dr. Price who offers me hands-on research experience, believes my ability to do research, and shares his wisdom about doing good research.

I would also like to thank Dr. Min Chi and Dr. Eric Wiebe for their encouragement and constructive feedback for my research.

Additionally, I am grateful to the co-authors who helped on this work: Yihuan Dong, Nicholas Lytle, Samiha Marwan, and Alexandra Milliken. I would also like to thank ToniAnn Marini and Rusty Martin, who helped conduct my research experiment in a classroom. I am thankful to work with other colleagues on their research as well: Ye Mao, Dr. Andrew Hicks, Dr. Zhongxiu (Aurora) Peddycord-Liu, and Dr. Veronica Cateté.

My thanks also go to other colleagues in the Game2Learn lab and the HINTS lab: Dr. Behrooz Mostafavi, Christa Cody, Rachel Harred, David Warren, Mehak Maniktala, Amy Isvik, Preya Shabrina, Michael Suggs, and Wengran Wang.

In addition, I am grateful to my internship mentors: Jesse Windle, Michal Cierniak, Zach Weiner, and Miorel Palii. I am also thankful for having the chance to assist Mr. Dana Lasher and Dr. Jon Doyle with their courses. I would also like to thank Kenneth Tran and Jonathan Reese for helping lead the introductory computing courses at SPARC@MSEN. Thanks also to other friends who make my Ph.D. life more colorful and interesting: Charles Haithcock, Jennifer Tsan, Di Chen, Jianfeng Chen, Tianpei Xia, Yao Lu, Haiyu Yao, Lihua Hao, Sang Lu, Linting Xue, Zhangqi Zha, Zexi Chen, Shitian Shen, Shuai Yang, Shaoliang Nie, Guojing Zhou, Ran Tan, Boxuan Zhong, Rong Huang, Yiqiao Xu, Xinyue Yuan, PengCheng Wang, Xiaoting Fu, Lin Dong, Zhentao Tong, Yuhe Liu, Peng Jin, Qing Guo, Tingsyue Wang, Lei Wang, and Jingjie Zheng.
Special thanks go to my mother, father, sister, and brother, who supported me to chase my
dream.

This thesis is based upon work supported by the National Science Foundation under grant
No.1623470: “EXP: Data-Driven Support for Novice Programmers”.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................. ix

**LIST OF FIGURES** ............................... x

**Chapter 1**  
Introduction ................................. 1
1.1 Motivation ..................................... 1
1.2 Research Questions ........................... 4
1.3 Contributions ................................. 4
1.4 Dissertation Overview ........................ 5

**Chapter 2**  
BACKGROUND AND RELATED WORK ............. 6
2.1 Novice Programming Environments .......... 6
2.1.1 Scratch .................................... 7
2.1.2 BOTS ..................................... 8
2.1.3 Gidget .................................... 8
2.1.4 Alice ..................................... 9
2.1.5 Looking Glass ............................... 10
2.1.6 Snap! and iSnap ........................... 11
2.2 Learning Theories ............................. 14
2.2.1 Cognitive Load Theory .................... 14
2.2.2 Cognitive Theory of Multimedia Learning . 16
2.2.3 The ICAP Framework ....................... 18
2.3 Instructional Supports ....................... 21
2.3.1 Instructional Text ........................ 21
2.3.2 Worked Examples .......................... 21
2.3.3 Self Explanation ........................... 23
2.3.4 Parsons Problems .......................... 24
2.3.5 Example-based Feedback ................. 26

**Chapter 3**  
Study 1: Instructional Text, Worked Examples, and Erroneous Worked Examples In Novice Programming Environment .............. 28
3.1 Introduction ................................... 29
3.2 Methods ..................................... 30
3.2.1 Instructional Support Design ............. 30
3.2.2 Content Design ........................... 32
3.3 Results ..................................... 33
3.4 Discussion ................................... 35
3.5 Conclusion ................................... 37

**Chapter 4**  
Study 2: Design and Evaluation of Incomplete Worked Examples with Self-explanation Prompts in Snap! ......................... 39
4.1 Introduction and Background ............... 40
4.2 Methods ..................................... 40
## Chapter 8  CONCLUSION

8.1 Research Questions 100
   8.1.1 Research Question 1 100
   8.1.2 Research Question 2 102
   8.1.3 Research Question 3 102

8.2 Contributions 102

8.3 Future Work 103

### BIBLIOGRAPHY

### APPENDICES

Appendix A Assignments 117
   A.1 Incomplete Worked Example with Self-explanation Prompts Study Assignments 117
      A.1.1 Daisy Design 117
      A.1.2 Spiral Polygon 119
      A.1.3 Brick Wall 122
   A.2 iSnap Assignments 123
      A.2.1 Polygon Maker 124
      A.2.2 Squiral 124
      A.2.3 Guessing Game 1 125
      A.2.4 Guessing Game 2 125
      A.2.5 Pong 1 125
      A.2.6 Pong 2 126

Appendix B Example-based Feedback Rating Rubric 127
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table 3.1</th>
<th>Interfaces for the Instructional Supports</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.2</td>
<td>Experimental Puzzle example with buggy code (buggy codes are highlighted and should be deleted.)</td>
<td>33</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Mean (SD) and median total experimental puzzle completion time in minutes</td>
<td>34</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Study Outline</td>
<td>43</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Mean (SD) pre-test, post-test1, and post-test2 scores</td>
<td>44</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Mean (SD) factor score of cognitive load (IL - Intrinsic Load, EL - Exogeneous Load, GL - Germaine Load)</td>
<td>45</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>The first-quartile (Q1) and median (Med) time (in minutes) taken by the WE and non-WE groups to complete 1, 2 and 3 additional objectives after the WE objectives, as well as the proportion that Comp. each objective</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Average time (with Standard Deviation) for accomplishing the assignment in different semesters.</td>
<td>59</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Average class grades (with Standard Deviation) for each assignment in different semesters.</td>
<td>60</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Average hint requests (with Standard Deviation) of students for homeworks. The percentage of students who request hints are shown with %.</td>
<td>60</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Expert Features and Corresponding Data-Driven Features, as derived in [Zhi18b].</td>
<td>73</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Total number of generated example pairs, corresponding average snapshot coverage and hint coverage in the expert-defined feature space.</td>
<td>80</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Mean attribute ratings (with standard deviation) for example pairs in from each approach.</td>
<td>80</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Instructional supports categorization based on ECLEP framework.</td>
<td>91</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1  BOTS game interface. Players drag and drop commands to program a robot to navigate a puzzle. ........................................ 9
Figure 2.2  Gidget interface (Source: http://www.helpgidget.org/). Students write code to help a robot called gidget accomplish certain tasks. ............................. 10
Figure 2.3  Looking Glass completion programming puzzle interface [Har15]. Students need to use the available commands (A) to write correct programs (B) to create correct animation (C). If students get stuck, they can get feedback (F) by comparing the correct animation with theirs (D). ................................ 11
Figure 2.4  Snap! interface (Source: https://snap.berkeley.edu/). Students can drag and drop blocks from the code palette (left) into a scripting edit area (center) to program. The visual output will be shown on the stage (top-right). .......... 12
Figure 2.5  An example of hint suggestion in iSnap. When a student requests help, the suggested next-step code will be shown in a popup window. ....................... 12
Figure 2.6  An example of hint suggestion in iSnap. When a student requests help, the magenta highlighted block needs to be removed, and the yellow highlighted blocks are not in the correct position and need to be moved. ......................... 13
Figure 2.7  A cognitive theory of multimedia learning [May02]. .......................... 17
Figure 2.8  The four aspects of the ICAP Framework [CW14]. ........................... 19
Figure 3.1  Boxplots of completion time (y-axis) on the first three assessment puzzles (x-axis) for each condition (color). ................................................ 35
Figure 3.2  Boxplots of solution code length for each condition for the first three assessment puzzles. .......................................................... 36
Figure 4.1  Peer Code Helper Interface. Students can run the example code (1), reflect on the output (3), and explain the code (2). Examples are shown side by side. Fill-in-blanks answer choices are marked correct in green or incorrect in red. 42
Figure 4.2  The average number of objectives completed by each group over time, with shading indicating ± 1 standard error. ............................... 46
Figure 5.1  Parsons problem interface in Snap!. Available code blocks are shown in the code palette. ................................................. 55
Figure 5.2  A hint example: yellow highlights misplaced blocks or variables, blue shows input that needs a variable. ............................... 55
Figure 5.3  A 2D visualization of the similarity between students’ snapshots at different points in time, across semesters, for PolygonMaker and Squirrel. Color indicates the snapshot’s semester, and black borders indicated a submitted solution. ......................... 64
Figure 6.1  Example-based feedback prototype in iSnap. ........................................ 69
Figure 6.2  Squirrel pseudocode, Snap! code and output [Zhi18b]. ...................... 71
Figure 6.3  DDF-IS and Expert examples for one snapshot (red/green indicate changed code in the example). ............................................. 83
1.1 Motivation

Learning programming is a complex task and can be very difficult for beginners. As Mason et al. argued [Mas16], “computer programming as a content domain has a high level of element interactivity\(^1\), and is thus difficult to learn.” Similarly, as Robins et al. pointed out, learning to program is hard and novices often perceive programming courses as difficult [Rob03]. Researchers have put decades of effort into understanding how people learn programming [May81; PK84; BS85; Kur86; Pea87; Ich17] and improving how students learn programming [Res09; Lee13; Har15; Eri17]. The ability to code, or to program, has become an important skill for ensuring computational literacy and workforce readiness. According to the projected growth of computer and information technology occupations, which is faster than the average job growth, there are expected to be about 557,100 new computing jobs in the United States by 2026 [BLS18].

To facilitate novice learning and lower the barrier to learning programming, novice programming environments (NPEs) have been developed and incorporated into introductory Computer Science (CS) curricula such as CS Principles\(^2\) [AB12] and Exploring CS\(^3\) [Goo12]. Most NPEs provide a block-based programming interface (e.g. Scratch [Res09], Snap! [Gar15], Looking Glass [Har15]), which allows novices to construct program by dragging and dropping instead of typing, to avoid syntax

\(^1\)i.e. Students need to consider multiple components simultaneously to understand a concept.
\(^2\)http://www.csprinciples.org
\(^3\)http://www.exploringcs.org
errors. The block-based feature of NPEs has been shown to improve programming efficiency [PB15] and learning [WW17]. Additionally, as Bau et al. point out, novices can build complex and delightful creations rapidly within block-based programming environments [Bau17]. Even though students may perceive block-based programming to be easier than traditional text-based programming [WW15a], students may still struggle to learn programming. For example, Ichinco et al. showed that novices struggle to use examples to solve block-based programming problems [Ich17]. An international survey about learning programming conducted by Lahtinen et al. show that students perceived programming to be difficult not only because of its abstract concepts, but also because of the issues related to program construction such as program design, decomposition, and debugging [Lah05]. The inherent difficulties of program construction are unlikely to be lessened by the block-based programming environment alone. Additionally, informed by the results from cognitive science and educational psychology, Guzdial questioned whether starting novice programmers off by writing programs is the best way for them to learn such a complex skill [Guz15].

One major challenge is that programming often involves complex cognitive processes that may overwhelm students [Mor15; VM03]. To address this, we can use Cognitive Load Theory (CLT) [Swe98] and the Cognitive Theory of Multimedia Learning (CTML) [May02] to inform our design of student learning materials. CLT assumes that the human brain has a limited amount of cognitive resources, or working memory, to process information at one time, believes that all learning activities impose a cognitive load on the working memory. CLT suggests that students’ learning can be impaired if instructional materials require students to hold excessive amounts of information in working memory. CTML makes similar assumptions to CLT and can be applied to the design of multimedia learning environments. CTML can be used to make concrete suggestions (e.g. pretraining, weeding, signaling, etc) are made to reduce cognitive load when designing instructions in multimedia learning environment [MM03], as discussed in more detail in Chapter 2. Similarly, based on CLT, Sweller argues that instructional support can be used to help reduce students’ cognitive load [Swe98]. There are many types of instructional supports including self-explanation [Ber09], worked examples [Swe06], and erroneous worked examples [DRJ12]. Prior work shows the effectiveness of worked examples in guiding students on problem solving in mathematics [SC85], propositional logic proofs [Liu16], and stoichiometry [McL14]. Durkin and Johnson’s work shows the effectiveness of erroneous worked examples to facilitate learning in mathematics [DRJ12]. When combined with worked examples, self-explanation can be especially beneficial to students’ learning [Ber09]. Unfortunately, successful findings from cognitive science does not always transfer to other contexts, such as programming. For example, Morrison et al.’s study shows that the modality of explanations for reducing students’ cognitive load, such as presenting a diagram while delivering the explanation of learning material through audio only instead of through text, what works in geometry and physics, does not work in computer science [Mor17]. Novice programming environments (NPEs) are already popular, and provide features to aid novices. However, only a few studies have been conducted to evaluate different
instructional supports in novice programming environments (e.g. [Har15], [Ich17]). Additionally, it is still unclear which instructional supports are effective for novice programmers, and how different instructional supports compare to writing code in novice programming environments.

Both CLT and CTML operate under the assumption that learners are cognitively engaged with the learning material (i.e. the active processing assumption in CTML [May02] and the assumption for germaine load in CLT [Swe10a]), but this assumption may not hold if the instructional support is not designed properly to encourage learners to engage. Fortunately, we can use the Interactive-Constructive-Active-Passive (ICAP) framework to further guide the instructional support design. The ICAP framework defines cognitive engagement activities based on student overt behaviors during a learning activity [CW14]. The engagement activities are orthogonal to instructional tasks and can be categorised into one of four modes: Interactive, Constructive, Active, and Passive. The degree of positive learning outcomes from the activity are predicted by the framework in this order: Interactive > Constructive > Active > Passive. However, based on CLT, a learning activity that overloads student cognitive capacity would impede learning. Therefore, we combine the insights of ICAP and CLT to consider both the learning outcome order of ICAP framework and student cognitive capacity. Therefore, the ideal instructional support is engaging and would increase students’ mental effort on digesting the learning material (called germaine load in CLT) without overloading their cognitive capacity. As Sanders et al. suggest [San17], we can design better techniques to “demand thinking.” Informed by the ICAP framework, and by integrating it with CLT and CTML, I believe we can design instructional supports to help novices learn programming more efficiently than writing code from scratch.

In this work I present the design and evaluation of different instructional supports for novice programming environments, informed by theories from educational psychology and cognitive science. The success of worked examples in other domains motivated me to design and evaluate them in programming. Specifically, I used CLT and CTML to design supports including: instructional text, worked examples, erroneous worked examples, incomplete worked examples with self-explanation prompts, Parsons problems, and adaptive, data-driven example-based feedback in novice programming environments. I conducted studies to evaluate each of these supports in empirical studies in real workshops and classrooms, except for example-based feedback, which uses an expert evaluation. Since practice is an important step for constructing schemas (related to building expert knowledge) and students still have to program at some point, to add in the benefits of instructional supports (e.g. worked examples) to problem solving and to advance my goal of the automatic creation of adaptive instructional supports for novices solving open-ended problems, I designed a new algorithmic, data-driven approach to derive adaptive example-based feedback. Finally, I developed a framework called ECLEP that will help guide the design of new instructional supports that can strike a balance between student engagement and effort, and provides insight into the effectiveness of diverse instructional supports.
1.2 Research Questions

The goal of my research is to develop insights into designing effective, theoretically-grounded instructional supports in the novice programming environment. To reach my research goal, in this work, I am interested in investigating the three high-level research questions:

• **RQ1**: How do different types of worked examples compare to problem solving in novice programming environments?

• **RQ2**: How do Parsons problems compare to problem solving in novice programming environments?

• **RQ3**: How can we use data to provide high-quality example-based feedback automatically?

**RQ1** is addressed in Chapters 3 and 4, which discuss the design and evaluation of worked example and its variants in two different NPEs. **RQ2** is addressed in Chapter 5, which presents a novel design of block-based Parsons problems and discusses how Parsons problems can save students time compared to writing code. Chapter 6 also addressed **RQ3**, which presents a novel method to generate example-based feedback from student data and discusses the quality of example-based feedback generated from different approaches based on expert evaluation and how to improve my algorithm. Overall, I designed and evaluated different instructional supports to gain a better understanding and insights on how to use theories to inform the instructional support design.

1.3 Contributions

This dissertation offers the following primary contributions:

1. The design and evaluation of instructional text, worked examples, and *erroneous* worked examples in a novice programming environment. From this study, I identified the tradeoffs in the design of instructional supports in terms of engagement, learning, and time spent on the learning activities.

2. The evaluation and design of incomplete worked example with self-explanation prompts in a novice programming environment. This study shows a promising direction for balancing the reduced cognitive load of worked examples with the increased activity engagement of problem solving in instructional support design.

3. A novel design of Parsons problems in a novice programming environment.
4. A comparison of the impact of block-based Parsons problems and problem solving on student learning efficiency. This study showed that Parsons problems can improve student learning efficiency by reducing the programming solution space and preventing unproductive behaviors.

5. A novel method for generating data-driven example-based feedback, which enables adaptive example-based feedback for students. This study demonstrates the potential for providing relevant, adaptive and high-quality example-based feedback to students.

6. A new framework that integrates the ICAP activity engagement framework with Cognitive Load Theory to explain the differences among instructional supports based on how well they promote engagement and effort while reducing cognitive load.

1.4 Dissertation Overview

The remainder of this dissertation presents the design and evaluation of a variety of instructional supports in four studies. Chapter 2 presents a review of relevant literature, including learning in novice programming environments, educational psychology theories and frameworks, including Cognitive Load Theory, ICAP activity engagement, and the cognitive theory of multimedia learning, and the impact of diverse instructional supports on learning. Chapter 3 presents results from a comparison of different instructional supports including instructional text, worked examples, and erroneous worked examples in a block-based programming game called BOTS. Chapter 4 details the design and evaluation of incomplete worked examples with scaffolded self-explanation prompts in a block-based programming environment called Snap!. Chapter 5 discusses the comparison between block-based Parsons problems and writing code from scratch in Snap!. Finally, to bring the benefits of instructional supports into problem solving and further our goal of the automatic creation of just-in-time, adaptive instructional supports for novices solving open-ended problems, Chapter 6 presents an algorithm for generating data-driven example-based feedback and its evaluation based on experts in Snap!. Chapter 7 introduces the Engagement, Cognitive Load, and Extent of Programming (ECLEP) framework, discusses the studies and recommends instructional support designs based on the framework. The final chapter summarizes this dissertation with a discussion of results on my research questions, the contributions of this work to the computer science education research community, and directions for future work.
In this work, I designed and evaluated different instructional supports in novice programming environments and used data to automatically generate a new, adaptive instructional support for problem solving. Therefore, in this chapter, I review previous work on novice programming environments and their empirical support. Then I present the theories from educational psychology and cognitive science that can inform designers about the potential impacts of different types of instructional supports. Afterwards, I review how these theories relate to instructional supports and how they may impact learning in different domains, especially in programming. To my knowledge, only a few existing novice programming environments have instructional support features, and only a few studies have explored the impact of instructional supports compared to problem solving in novice programming environments.

### 2.1 Novice Programming Environments

Novice programming environments (NPEs) are specifically designed to lower the barriers to learning programming, for example through graphical output, clear feedback, and simplified programming languages. They have become widely used in introductory Computer Science curricula (e.g. [AB12], [Goo12]). This section reviews several exemplary NPEs and highlights the common features they share. The majority of NPEs reviewed in this section also use a block-based programming interface, which allows novices to construct programs by dragging and dropping blocks from a code palette.
instead of typing. As Bau et al. point out, “Blocks environments improve learnability for novices by using direct manipulation of blocks to prevent errors and enhance understanding of program structure.” [Bau17]. Empirical studies have also shown that the block-based feature of NPEs can improve programming efficiency [PB15] and learning [WW17]. Price and Barnes compared the textual and block interfaces with middle school students in a block-based programming environment called Tiled Grace. They found students using the block interface completed more objectives of a programming assignment in less time than students using the textual interface [PB15]. Similarly, Weintrop and Wilensky conducted a quasi-experimental study to compare the impact of block-based and text-based programming environments in a high school programming class. The results of their study show that students using the block-based programming environment have greater learning gains and are more interested in future computing courses than students using the text-based programming environment [WW17].

2.1.1 Scratch

Scratch is a popular block-based programming language to teach children programming, with the goal to “nurture a new generation of creative, systematic thinkers comfortable using programming to express their ideas.” [Res09]. It is similar to the idea of the LOGO programming language, which was designed by Papert et al. to teach novices programming [Log15]. LOGO is a simplified programming language where students can write a program to control a turtle to draw things from lines and squares to complex patterns like spirals. Compared to other programming environments, Scratch is designed to be “more tinkerable, more meaningful, and more social” [Res09]. It also benefits students more than LOGO. A comparison study conducted by Lewis shows that students using Scratch perceived less difficulty in learning boolean logic ‘AND’ and learned conditionals better than students using LOGO [Lew10]. In Scratch, students can program by dragging and dropping commands into a script window, eliminating the need for them to learn programming syntax. Scratch also provides colored commands, animations, and a library of cartoon-like sprites, which allows students to make games, create animations, and draw colorful patterns by programming.

Scratch has been evaluated in different contexts and it is beneficial for students to learn programming. Meerbaum-Salant et al. designed and evaluated a semester-long Scratch curriculum to teach middle school students computer science concepts in regular classrooms in two schools. They found that the majority of students can learn important concepts such as loops and message passing but struggle in learning variables and concurrency [MS10]. Another classroom setting study shows that Scratch experience aids students in learning textual programming languages. Students who had learned Scratch before can learn new topics faster, encounter fewer learning difficulties, and understand loops better than those who do not have Scratch experience [Arm15]. Additionally, Rizvi et al.’s study suggests that Scratch can help students maintain a high degree of perceived
self-efficacy in a CS0 course [Riz11]. In addition to classroom settings, Scratch has been evaluated in informal settings. Maloney et al. analyzed 536 Scratch programs created by students in an urban after-school center, they found those novice programmers prefer to use Scratch than other media-creation software such as Adobe Photoshop and their created programs cover concepts such as loops, conditionals, boolean logic, and variables [Mal08].

2.1.2 BOTS

BOTS [Hic14; Liu17] is a web-based educational game that teaches 6-12th grade students problem-solving and basic programming concepts including loops and functions (shown in Figure 2.1). In this work, I investigated the effect of instructional text, worked examples, and erroneous worked examples in BOTS (more details in Chapter 3). Novices in both BOTS and LOGO learn programming by writing code to navigate a robot (in BOTS) or a turtle (in LOGO), and immediately receive visual feedback by seeing its movement. In BOTS, players must solve each puzzle by placing the robot or boxes on yellow target switches (Figure 2.1(5)). Players drag and drop commands including movements, loops, and functions (Figure 2.1(4)) to the programming area (Figure 2.1(3)) to control the robot. Players can execute their programs via clicking the ‘Play’ button or the ‘Step’ button (Figure 2.1(1)). If there are errors in the code, a message in red will pop up to explain the error (Figure 2.1(5)). When a level is completed, players earn either a platinum, gold, silver or bronze medals, based off of how few commands they used, with fewer commands earning higher level medals. The reward system (Figure 2.1(2)) shows the best score (least commands needed, shown as 11 with four stars in Figure 2.1(2)) in this puzzle and the player’s current score (current number of commands used, shown as 7 in Figure 2.1(2)). Most BOTS puzzles have repetitive patterns, which encourage players to use loops and/or functions to shorten their code to earn a better medal. BOTS has a leaderboard to engage competitive players [Hic14].

2.1.3 Gidget

Gidget is a puzzle-based debugging game, where students write Python-like code to control the movement of a robot called Gidget. For example, ‘left 2’ moves Gidget left for 2 steps. In this game, students need to fix existing code to control Gidget to finish certain tasks such as pick up an animal called piglet to a certain location (shown in Figure 2.2). The game teaches students programming by presenting faulty code (code with errors) and letting students find the error and fix it. As the example puzzle shown in Figure 2.2, some example code is already presented to the students and they can run the program at the beginning. In this puzzle, students need to program the gidget to grab the piglet and put it into a target location. To do that, students need to locate code which introduces bugs in the program and fix it. For example, in the Figure 2.2, one solution is to change ‘up’ to ‘up 3’, the second ‘left 2’ to ‘left 1’, delete ‘up 2’ and ‘right 6’, then add ‘down 2’ to accomplish the puzzle.
Lee et al. [Lee14] found that Gidget helps middle school students learn programming concepts, including conditionals and loops, within 5 hours. Similarly, another study conducted by Lee and Ko shows that students playing through the Gidget game learned similar concepts more efficiently than students learning from an online Python course [LK15a]. However, students who create their own puzzles in Gidget did not learn much, as indicated by the lack of learning gains from pre-test to post-test scores. This is one indication that the design of instructional support within a novice programming environment is critical in promoting learning.

### 2.1.4 Alice

Alice is a media-rich, 3D interactive programming environment, where novice programmers use menus to select commands to move 3D objects or change their physical appearance [Coo00]. Alice provides a library of sounds and 3D objects, which allow students to create 3D animations and is widely used in introductory programming classes. Like Scratch and BOTS, Alice has the block-based feature and provides immediate visual feedback by showing the movement or appearance of objects when the program runs. Alice was found to be more effective than textual programming languages in teaching students programming [Mos04; Bij13]. Moskal et al. integrated Alice into a CS1 curriculum for teaching novices fundamental object-oriented programming [Mos04]. They found the Alice
course is effective in improving students' grades, retention rate, and attitudes towards computing in CS1, compared with students in a traditional CS1 class. Similarly, in a quasi-experimental classroom study with 30 undergraduate students, Biju found that introduce Alice along with C++ is more effective in teaching programming than using C++ alone, as indicated by significantly higher quiz scores from students who also learned Alice than students who just learned C++ [Bij13]. In addition to effectiveness, Alice has also been found to be motivating and engaging. In an 8-day summer camp study, Adams taught middle school students Alice, covering concepts including variables, functions, lists, loops, conditionals, and events [Ada07]. The survey results indicate that students enjoyed and felt highly motivated to create animations using Alice. Similarly, Bishop-Clark et al. investigate the effect of Alice in an introductory computing class and they found that students enjoyed programming more, were more confident in programming, and understood programming concepts better (i.e. had significant learning gain from conceptual pre-test to poest-test) after studying Alice [BC07].

2.1.5 Looking Glass

Looking Glass [Har15] is an extension to Alice, which allows novices to author 3D animated stories using a block-based programming language. Students can drag and drop program statement tiles to control the movement of objects in the environment. The program will show 3D animations when executed. Except for step-by-step tutorials, Looking Glass also provides programming puzzles, which are similar to Parsons problems (introduced below in Section 2.3.4). Harms et al. have shown
that the programming puzzles is more effective in teaching novices than a step-by-step tutorial [Har15], as I discuss in more detail in Section 2.3.4 of this Chapter.

![Looking Glass completion programming puzzle interface](Har15)

Figure 2.3 Looking Glass completion programming puzzle interface [Har15]. Students need to use the available commands (A) to write correct programs (B) to create correct animation (C). If students get stuck, they can get feedback (F) by comparing the correct animation with theirs (D).

### 2.1.6 Snap! and iSnap

Snap! is a block-based programming environment, which is used extensively in the Beauty and Joy of Computing (BJC) AP Computer Science Principles curriculum [Gar15]. Similar to other environments, students can construct code to control movement of one or more Sprites. Students can also create animations, games, simulations or drawings. A unique and advanced feature of Snap! is that it allows students to create their own blocks (functions and procedures).

iSnap [Pri17d] is an augmented version of Snap! with detailed logging and on-demand, data-driven hint. It logs student interactions with the system such as code edits and click events (e.g. clicking a button or selecting an item from a menu), and students’ complete code snapshots after each edit. The on-demand hint in iSnap are generated from previous students’ solutions based on the current student’s code [Pri16]. In iSnap, when students request help, if there are available hints near a block, a hint bubble will be shown near the block. After clicking the hint bubble, a popup window will be shown including students’ current code and a next-step suggested code highlighted in green (as shown in Figure 2.5).  

Except for suggesting adding new blocks, the iSnap also provides another type of hint such as deletion or modification. For example, in Figure 2.6, the student probably need to remove the magenta highlighted block, and put the yellow highlighted blocks in the correct position.

iSnap has been evaluated in several contexts. In a classroom pilot study, Price et al. found that
Figure 2.4 Snap! interface (Source: https://snap.berkeley.edu/). Students can drag and drop blocks from the code palette (left) into a scripting edit area (center) to program. The visual output will be shown on the stage (top-right).

Figure 2.5 An example of hint suggestion in iSnap. When a student requests help, the suggested next-step code will be shown in a popup window.
over half (33/62) students request at least one hint, and nearly half (49.4%) of the hints were followed by students [Pri17d] and those hints did help student complete the assignment objectives. However, the pilot study also indicates that many students do not request hint at all. Another classroom study investigated this phenomenon and Price et al. found that the quality of first few hints is positively correlated with future hint use [Pri17c]. Even the hint is beneficial to students, they may deter to request help after receiving poor quality hints [Pri17c]. To further explore the hint quality, Price et al. found that the data-driven hints can suggest code edits almost as frequently as another human tutor, but it may also suggest many code edits that are not supported by human tutors [Pri17a]. Further, they evaluated the quality of data-driven hints with expert evaluation and found that the quality of hints generated by student data outperforms a single expert solution but not as good as a comprehensive set of expert solutions [Pri18]. Interestingly, with more student data, the hint quality may decrease. Except for investigating the quality of hints, to understand how and why student seek and avoid help, Price et al. did a qualitative study with 15 undergraduates and they found student have different expectations of human and computer tutors [Pri17b]. Their analysis suggests that computer tutor is more accessible and less threatening to students’ perceived independence than a human tutor. However, computer tutor is less trustworthy, interpretable, and perceptive than a human tutor [Pri17b].

This work is based on iSnap. Snap! is not only a popular programming environment, but also a web-based system and easy to access from. Additionally, the open source of Snap! also allows researchers to modify and expand it easily. Except for the benefits of Snap!, the logging feature in iSnap allows researchers to mining meaningful information from student code traces or to build data-driven features. Based on those reasons, I have selected iSnap as the target environment for
2.2 Learning Theories

This section introduces the theoretical foundations of this work, including Cognitive Load Theory [Swe98; Swe11a], Cognitive Theory of Multimedia Learning [May02], and the Interactive-Constructive-Active-Passive (ICAP) framework for activity engagement [CW14].

2.2.1 Cognitive Load Theory

Cognitive Load Theory (CLT) proposes a framework to describe how mental effort, or “cognitive load,” is utilized when processing information and problem solving, and how the distribution and amount of this load impacts learning [Swe98; Swe11a; Swe11b]. CLT assumes humans have limited processing capacity (working memory) and an unlimited information store (long-term memory) for schemas. A schema can consist of any knowledge or information that can be learned and stored in long-term memory. As our knowledge expands and skills develops, we can construct more complex, high-level schemas by combining many low-level schemas. A schema can be treated as a single element in working memory, no matter how complex it is and how much information it contains. What differentiates novices and experts is the amount of sophisticated domain-specific schemas in long-term memory. Compared to novices, experts have much more complex schemas in long-term memory that can be retrieved as a single element, allowing them to do more complex work that novices are not able to do. To construct schemas, students need to finish an important step named schema automation, which requires a lot of practice. With enough practice, students can retrieve a schema unconsciously and thus reduce the working memory load. For example, experts may understand a chunk of code snippets without consciously paying attention to the details such as syntax, while novices may need to consciously process the type of variables and the function of each code structures. The goal of learning is to construct complex domain-specific schemas which can be retrieved automatically.

When novices learn complex cognitive tasks, they may need to process multiple elements simultaneously and thus their working memory may become “overloaded” impeding learning. There are three types of cognitive load: intrinsic, extraneous, and germane. Intrinsic cognitive load results from the inherent difficulty of the learning material and cannot be reduced without omitting key elements of the material. Extraneous cognitive load is associated with the learning activity itself, including how information is presented and what actions the student must complete. Germane cognitive load is the mental effort expended in processing and learning new information. Based on CLT, both intrinsic load and extraneous load can be defined by element interactivity [Swe10a]. An element represents anything that can be or has been learned. Low element interactivity indicates
that the elements can be learned independently, such as learning the meaning of a word. While *high element interactivity* means the elements are highly interactive and cannot be learned in isolation. For example, when learning a for-loop in Java programming language, students need to process variables, arithmetic operators, assignments, and terminating conditions simultaneously. The level of interactivity is determined by the number of interacting elements which depends on both the nature of the information and knowledge of learners. The level of intrinsic load is assumed to be determined by the element interactivity of the learning material. The extraneous load depends on whether reducing the element interactivity would alter the learning content. The distinction between intrinsic load and extraneous load depends on “whether the element interactivity is essential to the task or whether it is a functional of instructional procedures.” [Swe10a]. Because the total cognitive load is determined by the intrinsic load and extraneous load, instructional designers are advised to design activities that minimize extraneous load to allow for germane load [Swe11c].

Empirical studies have found many instructional effects that are consistent with CLT [Swe11b]. The common CLT effects include the goal-free effect, the worked example effect, the problem completion effect, the split-attention effect, the modality effect, the redundancy effect, the expertise reversal effect, the guidance fading effect, the imagination and self-explanation effect, the element interactivity effect, the transient information effect, and the collective working memory effect. The goal-free effect indicates that students learn better by solving a problem with a non-specific goal rather than with a specific goal. The worked example and effect shows that students study worked examples perform better on post-test problems than students solve equivalent problems. The problem completion effect indicates that learners benefit more from solving partial completed problems than equivalent problems. The split-attention effect implies that students learn better from learning materials that integrate multiple sources of information together rather than those that have separated those sources either spatially or temporally. The modality effect indicates that students benefit more from learning instructions that present both visually and audibly, such as presenting animation along with narration rather than along with on-screen text. The redundancy effect indicates that students learn less from instructions with redundant information than from instructions without redundant information. The expertise reversal effect implies that expert learners may learn less from learning material which is essential for novices but is redundant for experts. The guidance fading effect indicates that guidance should be decreased gradually as learners become knowledgeable over time. The imagination effect implies that when students have sufficient expertise, they would learn better by imagining the problem solution procedure than studying the same worked example again. The self-explanation effect means that students benefit more by explaining and providing justification for worked examples than just studying the worked examples. The element interactivity effect is defined as that high intrinsic load is the premise of other CLT effects. We may not find any CLT effects if the intrinsic load is low and thus different instruction presentation would not overload students [Swe11b]. The transient information effect indicates that the information disappear and
students lose learning due to the lack of time for processing it or linking it with new information. The collective working memory effect means that students learn better by studying collaboratively than studying alone.

Based on CLT, many instructional supports have been designed with the intent to improve learning. Empirical studies have shown the effectiveness of applying CLT to the design of instructional supports to help novices learn complex cognitive tasks using worked examples [SC85; McL14]. My review of the literature showed that instructional text, worked examples, and erroneous examples were some of the more common support methods, and I discuss these in more detail in Section 2.3. I also discuss self-explanation prompts, which commonly accompany worked examples, and Parsons problems, an additional form of instructional support, which are specific to programming problems. Additionally, I introduce example-based feedback, which has elements both from worked examples and problem solving.

2.2.2 Cognitive Theory of Multimedia Learning

The Cognitive Theory of Multimedia Learning (CTML) describes how people learn from words and pictures in multimedia environments [May02]. CTML is based on three assumptions: dual channel, limited capacity, and active processing. Dual-channel assumes that people have two different channels in working memory to process materials: verbal and visual. The verbal channel focuses on processing the information come from the ears (i.e. audio information). The visual channel aims to process information received from the eyes. Limited Capacity assumes that both the verbal and visual channels have a limited amount of processing capacity to process information. Active processing assumes learners engage in active learning in the verbal and visual channels. The cognitive theory of multimedia learning is represented in Figure 2.7. The arrows indicate cognitive processing. The top row represents the verbal channel and the bottom row illustrates the visual channel. The arrow from words to ears represents spoken words as they are heard by learner’s ears and the arrow from words to eyes represents on-screen text as it is seen by learner’s eyes. Similarly, the arrow from pictures to eyes represents pictures as they are seen by learner’s eyes. In a multimedia learning environment, the learner should pay attention to the incoming spoken words (labeled selecting words in Figure 2.7) and the pictures (labeled selecting images in Figure 2.7), then mentally organize the selected words or pictures into coherent mental representations. Finally, the learner should integrate the new information with prior knowledge to achieve learning. As shown in Figure 2.7, this process is represented by combining the verbal model, pictorial model and learner’s prior knowledge. According to the CTML, the five relevant cognitive processes related to active learning include: selecting words, selecting images, organizing words, organizing images, and integrating. Mayer hypothesized that multimedia messages involving all the five of cognitive processes are more likely to generate meaningful learning than those that do not [May02].
Based on the CTML, there are three types of cognitive demands, which can be mapped to the three kinds of cognitive load in CLT. Essential processing is similar to germane load, which requires learners to cognitively process the learning material. Incidental processing corresponds to extraneous load, which comes from the design or presentation of the learning material. Representational holding is similar to intrinsic load, which indicates the cognitive process that learners use to hold the mental representation of the learning material in working memory over time. Due to the limited capacity of both the verbal and visual channels, the cognitive demands may overload these capacities and thus impede learning. Mayer proposed nine ways to reduce cognitive load in multimedia learning based on five cognitive overload scenarios [MM03]. The nine suggestions include off-loading, segmenting, pretaining, weeding, signaling, aligning, eliminating redundancy, synchronizing, and individualizing. Each suggestion leads to an instructional effect, which is described later. Off-loading suggests that when essential processing overload the visual channel, instruction designer to move some instructions from visual channel to auditory channel, which leads to the modality effect. Segmenting recommends instruction to be designed into segments and allow students to process each segment in their pace. Pretraining indicates having students to learn each component of instructions beforehand, which corresponds to the pretaining effect. Weeding suggests removing extraneous material that is not essential to learning, leading to the coherence effect. Signaling recommends adding cues to guide students attention on processing the learning material, which corresponds to the signaling effect. Aligning suggests to integrate pictures and words together to reduce incidental processing, which leads to the spatial contiguity effect. Eliminating redundancy advises avoiding presenting both on-screen and spoken text, leading to the redundancy effect. Synchronizing suggests presenting animation and narration at the same time, which corresponds to the temporal contiguity effect. Individualizing recommends to ensure learners are able to hold mental representations in memory, which maps to the spatial ability effect.

Similar to CLT, many studies have found effects that are consistent with the CTML [May02]. According to the CTML, common effects include multimedia effect, spatial contiguity effect, temporal
contiguity effect, coherence effect, modality effect, redundancy effect, pretraining effect, signaling effect, personalization effect, and spatial ability effect. The multimedia effect indicates that explaining the instructions in both words and pictures can improve students’ learning than presenting the instructions via words alone. The spatial contiguity effect is the same as the split-attention effect in CLT, showing that students learn better from learning materials that integrate pictures and words together rather than from those that separate pictures and words. The temporal contiguity effect shows that students benefit more from the instructions that present animations and narration simultaneously rather than successively. The coherence effect indicates that students learn more deeply from a concise presentation of learning materials rather than from an expanded presentation that includes non-essential information for learning. The modality effect is the same as the CLT modality effect, which implies that students learn better from a learning material which presents the animation along with narration rather than along with on-screen text. The redundancy effect is the same as CLT redundancy effect, both show that students benefit more from narrated animations that do not have redundant on-screen text. The pretraining effect indicates that students learn better when they have been trained to learn each component of the learning material than those who do not have such training. The signaling effect occurs when students learn more deeply from instructions that have signal cues to guide students’ cognitive processing during learning rather than from instructions that have no signals. The personalization effect indicates that students learn better from multimedia instruction that conveys the text in personalized conversational style rather than a formal textbook-like style. The spatial ability effect means that students with high spatial ability learn better from well-designed instruction than those with low spatial ability.

This work takes suggestions (e.g. segmenting, signaling) from CTML while designing the instructional support, as described in detail whenever the design of each instructional support is presented later in this thesis.

2.2.3 The ICAP Framework

This section introduces the Interactive-Constructive-Active-Passive (ICAP) framework, shown in Figure 2.8. The ICAP framework defines cognitive engagement activities based on student overt behaviors and categorizes the engagement activities into one of four modes: Interactive, Constructive, Active, and Passive [CW14].

The four modes of the ICAP framework are categorized by students’ overt behaviors during learning activities. The Passive mode is defined as students being oriented toward or receiving information from the learning material without doing any overt activities related to learning. For example, a student may just listen to a lecture or read a text without taking notes. The Active mode is defined as students taking some form of overt motoric action or engaging in physical manipulation during learning, such as taking notes while listening to a lecture or highlighting sentences while
Figure 2.8 The four aspects of the ICAP Framework [CW14].

reading a text. The Constructive mode is defined as students generating additional outputs or products beyond the learning material, such as self-explaining the concepts, or taking notes in their own words while reading a text. The Interactive mode is defined as dialogues that meet two criteria: (1) that both partners are primarily in constructive mode, and (2) that both partners should take turns sufficiently. For example, a student and a partner read a text and discuss comprehension questions.

In the ICAP framework, the taxonomy of the four modes (Interactive, Constructive, Active, Passive) requires eight assumptions including: content relevant, intended versus enacted, analyzing the outputs, advantages of externalized outputs, greater probability, independence, hierarchy, and intermode boundary. The content relevant assumption requires that students are doing activities related to the learning content. For example, when a student self-explain a worked example, the student’s explanation should be relevant to the example. The intended versus enacted assumption indicates that we should rely on students’ apparent behaviors as the measurement of engagement. The analyzing the outputs assumption means that we should analyze students’ response to confirm the actual mode of an activity. The advantages of externalized outputs assumption means that students’ external outputs (e.g. notes, diagrams) have practical, cognitive, learning, and epistemic advantages. The greater probability assumption defines the mode of students’ overt behaviors for a given learning activity by which mode the student is in for the majority of the time that they are actually engaging with learning materials. The independence assumption indicates that the engagement activities (e.g. taking nodes while listening to a lecture) are orthogonal to instructional tasks. In other words, it means the students’ behavior during learning is independent of the instruction. The hierarchy assumes that a higher mode subsumes a lower mode. For example, the Interactive mode
requires that both students are already in Constructive mode. The intermode boundary assumption means that the boundaries between two modes are not rigid. For example, problem solving is hard to classify and can fall on the boundary between two modes, depending on students’ cognitive processes.

The ICAP framework postulates four types of knowledge-change processes underlying each mode of engagement, and each process produces different changes in knowledge. The knowledge-change processes and their corresponding modes are: storing (Passive), integrating (Active), inferring (Constructive), and co-inferring (Interactive). Storing means students store the new information in an isolated way. Integrating indicates that students’ prior knowledge gets activated by the new information and the new information is integrated to prior knowledge while storing. Inferring means that not only are students’ prior knowledge and new information integrated, but also the integrated knowledge infers new knowledge. Co-inferring indicates that each student infers new knowledge from the integrated knowledge and also from the new information they gain by talking with partners.

In addition to the assumption that each mode has a specific type of knowledge-change process, the knowledge-change process has three other assumptions including: distinct from learning process, same set of knowledge-change process, and cognitive outcomes resulting from the knowledge-change process. The distinct from learning process assumption requires that the knowledge-change processes to be distinct from the cognitive processes of learning of a specific instruction or task. The same set of knowledge-change process assumes that the same mode of engagement contains the same underlying knowledge-change processes, even though students’ overt behaviors may be different. The cognitive outcomes resulting from the knowledge-change process assumes that different knowledge-change processes lead to different cognitive outcomes such as being able to recall, apply, transfer, and co-create. Recall indicates that students can verbatim recall the knowledge in an identical context. For example, students can reuse the same procedure for solving identical problems. Apply means that students can apply the knowledge to similar but different contexts (i.e. similar problems or concepts). Transfer implies that students can apply the knowledge to a novel context or distant problem. Co-create indicates that students can invent new explanations, procedures, ideas, and products from successful dialogue with partners.

Based on the knowledge-change processes underlying each mode of engagement, the ICAP framework hypothesizes that learning outcomes from each mode follow in this order: Passive < Active < Constructive < Interactive [CW14]. The corresponding learning outcomes of Passive, Active, Constructive, and Interactive are assumed to be mapped to different levels of learning, such as minimal understanding, shallow understanding, deep understanding, and deepest understanding. When Chi and Wylie proposed the ICAP framework, they also listed several empirical studies to support this hypothesis [CW14]. For example, Menekse et al. conducted a study to test the ICAP hypothesis and found that students’ learning gains increase significantly across Passive, Active, Constructive, and Interactive [Men13].
Both CLT and CTML assume that learners are cognitively engaged with the learning material (i.e. the active processing assumption in CTML [May02] and the assumption for germane load in CLT [Swe10a]). However, this assumption may not hold if the instructional support is not designed properly to encourage learners to engage. Therefore, to further guide the instructional support design, this work integrates the ICAP framework and Cognitive Load Theory to derive a new framework, as described in Chapter 7.

2.3 Instructional Supports

In this work, I embed instructional supports in the novice programming environment instead of separating them from the environment. Prior studies suggest that interleaving worked examples and problem solving is more beneficial to students than only solving equivalent problems (as described in more detail below) [TR93]. Additionally, separating the instructional supports and the programming environment may introduce split-attention effect, which may impose extraneous cognitive load and impair the effect of instructional supports since students need to split their attention between multiple sources of information and need to mentally integrated them [Swe11a]. Furthermore, Sweller et al. argued that it should be beneficial to assist novices by scaffolding during problem solving [Swe11a]. It is also unlikely that students will leave the programming environment to attend to examples or other instructional supports and then to return to the environment.

2.3.1 Instructional Text

Instructional text, also called instructional explanation, provides basic knowledge to solve a problem. For example, in Looking Glass [Har15], when a student first works in a tutorial condition, instructional text with videos of each step appear in a popup window to teach new code statement and guide the students to solve a puzzle. Similarly, Gidget [Lee13] provides an in-game guide and instructions to inform new players of the game mechanics. In an overview of the effectiveness of instructional text in worked examples, Witter and Renkl [WR10] found that instructional text along with worked examples are more beneficial for acquiring conceptual knowledge than procedural knowledge. According to the personalization effect in CTML, the instructional text are suggested to be designed in personalized conversational style.

2.3.2 Worked Examples

A worked example (WE) teaches students how to solve a problem by directly presenting a solution for the problem. It is an effective instructional strategy to help novices learn complex tasks and has been studied for decades [Swe06]. Prior studies have explored the effect of WEs in different disciplines, including algebra [SC85], propositional logic proofs [Liu16], and stoichiometry [McL14].
Empirical studies have shown that WEs are more beneficial to novice learners compared with traditional problem solving in learning algebra [SC85]. Sweller and Cooper found that students who were presented with WE problems later solved similar problems quicker and with fewer errors than students who solved the same problems themselves [SC85]. McLaren et al. compared four different instructional designs in a stoichiometry tutor. They found that students in the WE condition performed similarly in the posttest but were more efficient in finishing intervention problems and with less cognitive load, compared to students in other conditions including problem solving, erroneous worked examples, and tutored problem solving [McL14]. In an intelligent tutoring system for teaching and practicing propositional logic problems in discrete math, Liu et al.’s [Liu16] work shows that WEs are beneficial to undergraduate students in the first few problems they solve but less beneficial for later problems in the students’ homework set.

Alternatively, empirical studies have shown that “modular WEs” can also be designed to reduce intrinsic cognitive load. While WEs traditionally present the entire solution to a problem, modular WEs, “break down complex solutions into meaningful solution elements” [Ger06]. Gerjets et al. explored the effectiveness of modular examples in teaching probability and showed that modular WEs are more efficient and effective than traditional WEs and impose less intrinsic cognitive load on learners [Ger04].

2.3.2.1 Worked Examples in programming

Worked examples have also been studied in the domain of programming. Some studies suggest using worked examples in programming for teaching novices. Pirolli and Anderson investigated how three novices learn recursive functions and found that they relied heavily on examples while writing new programs [PA85]. Informed by the results from cognitive science and educational psychology, Caspersen and Bennedsen proposed a model which treats WEs as one of the fundamental principles of programming education [CB07]. Vihavainen et al. extended Caspersen and Bennedsen’s work and proposed a model, which also suggests using WEs in study materials and lectures [Vih11].

Others have investigated the impact of WEs experimentally. Trafton and Reiser’s study suggests interleaving WEs with similar practice problems to maximize students learning gains, compared to blocking WEs with problems, or solving equivalent problems by writing LISP programs [TR93]. Recently, Ericson et al. found that interleaving WEs with Parsons problems can improve programming learning efficiency compared to WEs paired with writing code or fixing code [Eri17]. Patitsas et al. investigated the effect of comparing and contrasting multiple solutions, by presenting the different solutions (WEs) side by side. They found that students in comparing and contrasting condition showed greater learning gains for procedural knowledge and flexibility than students in the WE condition [Pat13]. However, to our knowledge, only Trafton and Reiser directly evaluate the effectiveness of WEs by comparing them to equivalent problem solving tasks, but their results may
not apply to novice programming environment.

In novice programming environment, Ichinco et al. showed that novices struggle to use WEs to solve block-based programming problems [Ich17]. For example, they found that novices have difficulty in understanding the examples and finding relevant blocks to finish the programming task. Merriënboer and Croock found that incomplete WEs, in which students are given part of a WE and asked to complete the rest, improved novice's programming performance and post-test scores, compared with those who only had the WEs as a reference [VMDC92].

### 2.3.2.2 Erroneous Worked Examples

Erroneous worked examples are similar to worked examples but they include incorrect steps and require students to understand the information provided by the example, find the errors, and then fix them. Adams et al. explored the effect of erroneous worked examples in helping middle school students learn decimals via a computer-based tutor [Ada12]. They found no significant difference in students' immediate posttest performance between those in the erroneous worked examples and problem solving condition, but students in the former condition performed better on a delayed posttest. Some researchers have explored the effect of combining worked examples and erroneous worked examples. Große and Renkl's explored the efficacy of erroneous worked examples in helping students solve probability problems and they found that a mixture of worked examples and erroneous worked examples are more beneficial to students who have prior knowledge, compared to worked examples only [GR07]. In programming, erroneous worked examples are similar to debugging almost-correct code with small errors. The potential effectiveness of Parson's Programming Puzzles [PH06] and the success of Gidget [Lee13] shows that debugging may be a good way for teaching novice programming.

### 2.3.3 Self Explanation

Self-explanation is an instructional strategy which requires the learner to reflect and generate explanations about the learning material. It has been shown that combining self-explanation with WEs can be especially beneficial to students’ learning [Ber09]. A review of self-explanation shows that along with WEs, self-explanation prompts help students foster greater understanding [RC05] and improve students' problem solving performance [Mor15]. Morrison et al. show that students who self-explained subgoal labels perform better than those who were not given subgoal labels [Mor15]. Self-explanation has different forms, including open-ended, menu-based, focused, scaffolded, and resource-based self-explanations [WC14]. Berthold found that scaffolded self-explanation prompts following WEs were more beneficial to students in terms of conceptual and procedural understanding than open-ended self-explanation prompts in multimedia environments [Ber09]. Informed by the results from those studies, in Study 2 (Chapter 4), I implemented the scaffolded self-explanation
prompts along with worked examples in Snap! to guide students to reflect on example code.

In this work, Chapter 3 introduces the design and evaluation of instructional text, worked examples, and erroneous worked examples in BOTS. Chapter 4 presents the design and evaluation of incomplete worked examples along with self-explanation prompts in Snap!

2.3.4 Parsons Problems

Based on Cognitive Load Theory, instructional designers are advised to design activities that minimize extraneous load, extra elements that distract from the important learning material to allow for germane load [Swe98]. Merriënboer and Croock found that incomplete computer programs, in which students are given part of a correct program and asked to complete the rest, improved novice’s programming performance and post-test scores, compared with those who write code with examples as a reference [VMDC92].

Parsons problems are code-completion problems, in which students re-arrange mixed up code blocks into the correct order. Ericson et al. argued that Parsons problems have a more constrained problem space, and would impose lower cognitive load than programming from scratch [Eri17; Eri18]. In their original implementation, Parsons and Haden presented students with problem descriptions and a set of drag-and-drop code fragments [PH06]. Each code fragment comprises single or multiple lines and the code fragments may have incorrect code. Students select the correct code segments and put them in the right order to solve a puzzle. Parson and Haden argued that Parsons problems are more engaging than writing code, and their initial evaluation showed that the majority of students (14/17) thought this kind of problem is useful or very useful for learning Pascal [PH06]. Fabic et al. found Parsons problems are only effective for novices, and they would be more effective for supporting learning with self-explanation prompts [Fab17]. Some studies have shown that Parsons problems are engaging [Gar07; Eri15; Eri16]. Garner’s semester-long study with eight students revealed that students were generally engaged with Parsons problems [Gar07]. Ericson et al. integrated Parsons problems into an ebook and found that Parsons problems were engaging, as evidenced by more students attempted Parsons problems than multiple choice questions [Eri15], and were more frequently mentioned by teachers as being valuable than other questions [Eri16].

Parsons problems difficulty can be varied by changing what students are required to do. 2D Parsons problems are designed to teach Python, and require students to both reorder and indent the code correctly [IK11]. Ihantola et al. designed mobile phone-based 2D Parsons problems called MobileParsons, which allows users to make small edits to pieces, such as selecting the condition of an if statement [Iha13], but its effectiveness has not yet been evaluated. Helminen et al. analyzed students interaction traces and identified several common patterns and difficulties that novices face while solving 2D Parsons problems, showing students have trouble indenting the code and students go back to a previous code state [Hel12; Hel13]. Parsons problems can also include distractors,
incorrect code with syntax or semantic errors, designed to highlight common errors that novices make. For example, a distractor might use an undefined variable called ‘Total’ but the defined variable is ‘total’. Parsons problems are harder with distractors [Gar07; Den08; Har16]. Parson and Haden’s pilot study showing students perceived Parsons problem with distractors are helpful [PH06]. However, Harms et al. found that Parsons problems with distractors decrease learning efficiency compared to Parsons problem without distractors in a block-based programming environment [Har16]. Therefore, I did not incorporate any distractor in Parsons problems used in this study.

Moreover, Parsons problems have been studied as assessments. Studies have shown that solving Parsons problems correlates with writing code [Den08; CH17]. Denny el al. explored Parsons problems as CS1 exam questions and they found a moderate correlation ($\rho = 0.53$) between scores on Parsons problems and writing code [Den08]. Similarly, Cheng and Harrington found a strong correlation ($\rho = 0.65$) between the students’ score on the writing code problem and the Parsons problems [CH17]. Morrison et al.’s study shows that the Parsons problems can be more sensitive for assessing students’ learning gains than writing code [Mor16]. They found that Parsons problems, which allows students to show their understanding of the meaning and sequence of programs, can measure students’ knowledge that cannot be measured by writing code alone.

While it is hypothesized that solving Parsons problems is more efficient and effective than traditional problem solving, only a few studies have explored the effectiveness for Parsons problem compared to writing code in teaching programming [Eri17; Eri18; Har15]. Ericson et al. did a study that interleaved worked examples with Parsons problems, writing equivalent code, or fixing buggy code separately, and they found that when interleaving with worked examples, Parsons problems are more efficient in helping novices learn programming than writing equivalent code or fixing buggy code [Eri17]. Compared with writing code or fixing buggy code, Parsons problems save students time by providing the code and limiting students’ solution space. Similarly, Harms et al. showed that Parsons problems are more efficient and effective in helping middle school students learn block-based programming independently than a tutorial [Har15]. The tutorial provides students video and textual instructions, which requires students to read and follow the instructions by finding and putting the exact code blocks into the programming area. Then students can run the code and see the animation generated by their code. In Harms et al.’s study, students using Parsons problems spent 23% less time on training and performed 26% better on transfer tasks, while they had higher cognitive load than students in the tutorial condition [Har15].

Overall this research and others have shown that Parsons problems can be effective, especially for learning efficiency. However more research is needed to build a better understanding of how and why they are effective. We could support novices both effectively and adaptively with a better understanding of Parsons problems. For example, we can automatically create Parsons problems from examples if they are proven to be more effective than writing code. In Chapter 5, I integrate Parsons problems into block-based programming environment to evaluate their effectiveness when
compared to writing blocks-based code.

2.3.5 Example-based Feedback

While instructional support can provide helpful alternatives to traditional problem solving, programming practice problems still comprise the majority of student work in many programming classes. Therefore, another challenge is to adapt instructional supports for use while students are programming. Example-based feedback builds on the idea of a worked example, showing a correct piece of code to help students learn to program [Gro14b], but it is shown during problem solving, similar to a high-level on-demand hint, and may contain only a partial solution. As Gross et al. argue, showing relevant partial solution could impose less cognitive load to novices than showing a full solution [Gro14b]. Additionally, the partial solution may make it easier to present than an entire solution for complex programming problems. Research has shown that example code can improve student code solutions [Gro14b; HB11]. The international survey conducted by Lahtinen et al. also shows that both students and teachers think example programs are the most helpful materials to help learn programming [Lah05]. Additionally, Keuning et al. argue for the need for such feedback in their review on automated feedback generation for programming, saying “the very low percentage of tools that give code examples based on the student’s actions is unfortunate, because studying examples has proven to be an effective way of learning” [Keu18]. However, existing example-based feedback systems may rely on a library of expert examples [HB17; IK15; Coe17], which can be costly to maintain, as instructors are unlikely to create new examples [HB11]. Additionally, the example code usually shows a related problem [HB17] that requires students to transfer knowledge to solve their current problem, which may be challenging for weaker students who need more help.

Brusilovsky proposed a system called WebEx [Bru01], which can provide programming example code with explanations for students. WebEx allows teachers to add explanations for each line of example code and students can select to view these available explanations while learning from the example code. Later Yudelson and Brusilovsky improved the WebEx by adding adaptive navigation features, and they evaluated the system in a classroom setting and they found the system increases student motivation to learn non-mandatory materials [YB05]. Hsiao et al. built a new system called AnnotEx, which is based on the WebEx and allows students to annotate code examples line by line [HB11]. They found by incorporating community feedback into the system, weaker students will be benefit from the peer reviews, as evidenced by increase in quality of annotations [HB11]. However, those example-based feedback systems may rely on a library of expert examples [HB17; IK15; Coe17], which can be costly to maintain, as instructors are unlikely to create new examples [HB11].

Ichinco et al. explored how novices use example to solve problems and they found that novices struggle to use examples [IK15]. They found that novices have trouble connecting example to their own code. Even for novices who recognized related parts between examples and their code,
they may have trouble in finding the code blocks or having misconception about using the code blocks correctly. Recently, they did a follow-up study confirmed that novices may have trouble understanding the example, finding code blocks or having misconceptions in using the example code in block-based programming environment [Ich17].

Gross et al. proposed a data-driven algorithm to provide example-based feedback based on structured solution spaces and their pilot study shows the feedback is helpful to students and can improve students’ code overtime [Gro14a]. In another work, they proposed different strategies to select example-based feedback from either expert-authored examples or student solutions[Gro14b]. They found the examples derived from expert solutions were better than examples derived from student solutions. More specifically, compared to those derived from student solutions, students rated expert-authored examples as more helpful and made more solution improvements using examples derived from expert-authored solutions.

In this work, Chapter 6 presents an algorithm to generate data-driven example-based feedback in Snap! and the expert evaluations of feedback generated from different approaches.
This chapter was adapted from: Zhi, R. et al. “Exploring Instructional Support Design in an Educational Game for K-12 Computing Education.” Proceedings of the 49th ACM Technical Symposium on Computer Science Education. 2018. [Zhi18a]

The original text has been modified as follows: the Related Work section has been removed and integrated into Section 2.3 of Chapter 2, and the section describing BOTS has been removed, since this is covered in Section 2.1.2 of Chapter 2. Some design descriptions and study results are updated to support discussion of the ECLEP framework later in Chapter 7.

This chapter presents my work on designing and evaluating the instructional support design including instructional text, worked examples, and erroneous worked examples in an block-based programming game called BOTS. RQ1 is investigated in this Chapter and Chapter 4.
Abstract

Instructional supports (Supports) help students learn more effectively in intelligent tutoring systems and gamified educational environments. However, the implementation and success of Supports vary by environment. We explored Support design in an educational programming game, BOTS, implementing three different strategies: instructional text (Text), worked examples (Examples) and buggy code (Bugs). These strategies are adapted from promising Supports in other domains and motivated by established educational theory. We evaluated our Supports through a pilot study with middle school students. Our results suggest Bugs may be a promising strategy, as demonstrated by the lower completion time and solution code length in assessment puzzles. We end reflecting on our design decisions providing recommendations for future iterations. Our motivations, design process, and study’s results provide insight into the design of Supports for programming games.

3.1 Introduction

Educational games are an effective way to engage students in learning to program. Gee argues that games can be used to teach skills and enhance learning [Gee03], and empirical evidence shows that educational games can teach programming concepts [LK15b], and improve motivation and promote knowledge in topics like computer memory [Pap09]. However, as Clark and colleagues note, educational games research must move beyond efficacy studies to “explor[e] how theoretically driven design decisions influence situated learning outcomes” [Cla16]. In this work, we explore how Cognitive Load Theory [Swe98] can inform the design of instructional support strategies for a programming game.

Sweller’s Cognitive Load Theory (CLT) [Swe98] suggests that when a student’s cognitive resources are overloaded, learning becomes difficult. To address this, educators create instructional supports (Supports): pedagogical resources and activities that scaffold the learning process and help reduce cognitive load [Paa03]. Many educational games use tutorials with instructional text to help players learn the core game mechanics. However, in other educational domains, researchers have designed effective alternatives. For instance, a worked example shows a solution and explains how to solve a problem step-by-step. Prior work shows that worked examples improve students' learning efficiency in a variety of domains [SC85; McL14]. Similarly, erroneous worked examples show problems with minor errors in the solution and require students to fix these problems [Ada12].

In this paper, we explore how Supports can be implemented in a programming game called BOTS [Hic14; Liu17]. We design three Supports for BOTS: instructional text, worked examples, and erroneous worked examples (buggy code or Bugs). We motivate our Supports by comparing them with other implementations and grounding decisions in CLT. We evaluate our Supports in BOTS by conducting a controlled between-subjects pilot study. We compare participants’ in-game
performance under each Support by their puzzle completion time and solution code length. Based on previous results in comparing Supports [McL14], we hypothesize that (1) our Examples condition will complete the learning activities faster and (2) there will be no significant difference in final performance evaluations between conditions. We reflect on the results and our design decisions and recommend future design directions.

3.2 Methods

We designed three instructional supports (Supports) for a programming game called BOTS: instructional text (Text), worked examples (Examples), and buggy code (Bugs). We based our designs on how each Support was implemented in other domains, and adapted it to work in BOTS. We developed a series of levels and a change to our loop command to teach loops and functions using our Supports.

BOTS has three types of commands: action commands that control the movement of the robot similar to other LOGO-like environments, control commands like conditions and loops, and function commands that allow players to define custom functions. In previous version of BOTS, a loop was similar to the for-loop in Java. Players had to set a variable, create a condition check, and increase the variable by some number to form a loop, each mapping to a different command. This design requires students to understand all three commands and their interactions. Liu et al. found that students struggle to use loops in this implementation of BOTS [Liu17], which may be due to the high cognitive load induced by these separate loop components. We designed a new loop command, “Repeat Times”, which only uses one command. Students can input the amount of times that they want to run the body of the loop without needing to understand variables or incrementing them.

3.2.1 Instructional Support Design

Text provides students explanatory text on how to use the interface and the programming commands, as well as how to approach solving specific puzzles. Ginns et al.’s review shows that Text in a conversational style contributes to better learning compared with a formal style [Gin13]. We therefore adopted this conversational style by using second-person language, (e.g. “The ‘Move Forward’ Command helps you move your Fire Bot one step forward.”). Our Text was augmented with the Stencil technique [KP05], in which the game directs players’ attention to a relevant UI component by surrounding it with a translucent colored stencil. This technique was shown to be more efficient than a traditional paper-based tutorial [KP05]. This is also suggested by the Cognitive Theory of Multimedia Learning [MM03], known as signaling (described in Section 2.2.2 of Chapter 2). At the beginning of each BOTS puzzle, a window appears with instructions about how to use the corresponding commands correctly in this puzzle. Players can read through these instructions,
Table 3.1 Interfaces for the Instructional Supports

<table>
<thead>
<tr>
<th>Instructional Text (Text)</th>
<th>Worked Examples (Examples)</th>
<th>Buggy Code (Bugs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A popup window explains the commands in detail. A stencil hole highlights important components.</td>
<td>Expert solution shown in the programming area. A popup window explains the solution in detail.</td>
<td>Buggy code is shown in the programming area, players can delete the buggy code (3) and add new code (2).</td>
</tr>
</tbody>
</table>

follow the instructions and apply the commands introduced to solve the puzzle. While the Text condition used these messages to convey all instructional content, all three conditions used them to explain instructions at the start of a level. In both the Examples and Bugs conditions, the explanatory text explains the game mechanics and covers the same material (explanation for the loop and function commands) as in the Text condition. As each of the three conditions use explanatory text, we treat the Text condition as our control.

Examples, one of the most common Supports, teach students to solve a problem by presenting the solution. Some previous work suggests interleaving Examples with traditional problem solving for best results [SC85; Paa92], though more recent work suggests that Examples and Examples with problem solving may be equally effective [VG11]. We chose to explore the effects of Examples only, without interleaved problem solving. In the Examples condition, we give students an expert solution, which is an optimal program to solve the puzzle. Similar to the Text interface, we used a popup window with Stencils to guide students and explain the optimal solution. The Examples program is visible in the programming area and is not editable. The system walks through the code explaining it at a high level. After running the code, a question related to the puzzle is presented (e.g. “Can you write down how many boxes I need to pick up in this level?”). We include these questions to check whether players understand the Example, since if no questions are presented, many students may skip through the Example without paying attention [Bak08].

Erroneous worked examples are similar to Examples. They provide information for solving a problem, but they have some incorrect steps that require students to find and fix them. In the Bugs condition, the game presents students with an expert solution with 1 or 2 errors and asks
them to fix the code to complete the puzzle. A previous study shows that students enjoy debugging in BOTS [Liu17]. In the Bugs condition, code with errors will show up in the programming area with a window explaining the debugging mechanics and the puzzle’s code. Players can debug the program using five actions: delete buggy code (Table 3.1-Bugs-(3)), undelete buggy code, add new code (Table 3.1-Bugs-(2)), remove new code, and run code. The new code inserted by students is highlighted (Table 3.1-Bugs-(2)) to differentiate it from the original code. When the student deletes an original command, it is shown as being crossed out, but if a student removes some of their own new commands, those disappear. The design distinguishes the students’ code from the original code, which makes it easier for them to see the changes they have made. We intended to help students focus on solving the problem in the original buggy code rather than building a new solution themselves.

### 3.2.2 Content Design

We designed 4 sets of puzzles: 4 tutorial, 1 pretest, 5 experimental and 6 assessment puzzles (APs). The tutorial puzzles teach students the basic game mechanics of BOTS. The pretest puzzle evaluates students’ prior knowledge of loops or functions. Experimental puzzles teach loops and functions using one of the three Supports. The final APs are similar to the experimental puzzles, and test player’s ability to use loops and functions without Supports.

We designed 4 tutorial puzzles that teach players the basic game mechanics for moving the robot and interacting with blocks. We designed these puzzles using suggestions from prior work on building tutorials for learning systems by Shannon et al. [Sha13] and Mayer and Moreno [MM03]. Specifically, we used the segmenting suggestion from Mayer and Moreno [MM03], as described in Section 2.2.2 of Chapter 2. We did this through segmenting the concepts and disabling unnecessary commands (thereby reducing player’s extraneous cognitive load) and teaching one or two concepts (in this case, commands) at a time. We also designed puzzles that review commands learned in previous puzzles and praised effort on puzzle completion. We also adapted the Stencil technique into the tutorial [KP05].

We designed 5 experimental puzzles to teach loops, functions and their combinations. All participants were given the same puzzles with a different Support (Text, Examples, or Bugs). Table 3.2 shows one experimental puzzle with the Bugs Support.

To teach loops and functions, we included multiple types of errors in the Bugs condition. This includes placing action commands which are suppose to be in the loop body before the loop, adding extra action commands within the function command, and removing necessary action commands.

We designed 6 APs with increasing difficulty to evaluate the participants’ in-game performance. AP1 tests the participants’ knowledge about the basic game mechanics (e.g. movement, climbing). AP2-6 are designed to be similar to the experimental puzzles in their optimal solution structure.
Table 3.2 Experimental Puzzle example with buggy code (buggy codes are highlighted and should be deleted.)

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>Buggy Code (Bugs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fire Bot PROGRAM:</td>
<td></td>
</tr>
<tr>
<td>-Move Forward</td>
<td></td>
</tr>
<tr>
<td>Repeat Times : 6</td>
<td></td>
</tr>
<tr>
<td>Move Forward</td>
<td></td>
</tr>
<tr>
<td>Move Forward</td>
<td></td>
</tr>
<tr>
<td>Turn Left</td>
<td></td>
</tr>
<tr>
<td>Climb Up</td>
<td></td>
</tr>
<tr>
<td>End Repeat Times</td>
<td></td>
</tr>
</tbody>
</table>

AP2-3 test the loop command, AP4-5 test the function command, and AP6 tests for a combination of loops and functions.

3.3 Results

We conducted a pilot study with 21 middle school students (15 male, 6 female) to evaluate how students used Supports in a BOTS lesson. Each participant played a version of BOTS with one of the three Supports: Text (which served as a control), Bugs, or Examples. Players played a set of intro puzzles, a pretest puzzle, experimental puzzles, and assessment puzzles (APs). Participants also took a pre and post survey with questions about their interest and self-efficacy in computer science, but there were no significant differences between conditions or between pre and post. Our hypotheses for this study based off results from prior work [McL14] were (1) the Examples condition would complete the experimental puzzles faster than the other two conditions and that (2) there would be no differences in final AP performance between the three conditions.

Only the first 3 APs were completed by all participants, so our analysis is on these puzzles, which focus on basic movement commands (AP1) and loops (AP2-3). Three students demonstrated clear knowledge of loops on the pretest, so their data were excluded from analysis, leaving 5 Bugs, 6 Examples, and 7 Text participants. To evaluate the effectiveness of the supports, we measured participants’ performance on the APs by their completion time and solution code length. Low completion time reflects that players can find the pattern within a puzzle quickly and solve the puzzle efficiently. Shorter (closer to optimal) solution code length implies a better understanding of how to use advanced commands (loops) effectively. In addition, we measured the completion time of the experimental puzzles to evaluate the time efficiency of the Supports. Since both time and code length data are not normally distributed, we used Kruskal-Wallis test for all measures.
Table 4 shows the mean and median times for completing the experimental puzzles. We found a significant difference between conditions for the total time spent on the experimental puzzles ($H(2) = 6.475$, $p < .05$). Individuals in the Examples condition completed the set of experimental puzzles significantly faster than the other two conditions (Text vs. Examples ($W = 42$, $p < .05$); Bugs vs. Examples ($W = 30$, $p < .05$)). However, no significant difference was found in completion times between Bugs and Text ($W = 8$, $p < .149$). Code length was not compared on the experimental puzzles, since students in the Examples and Bugs conditions were guided to the optimal solution by the instructional mechanics. We also manually investigated students experimental puzzle solutions and found that $19/25$ (76%), $13/30$ (43.3%), $12/35$ (34.2%) of experimental puzzles in the Bugs, Examples and Text conditions, respectively, have been completed as expected (bugs were fixed, questions were answered correctly, advanced commands introduced in instructions were applied correctly). Additionally, there were $4/5$ (80%), $2/6$ (33.3%), $3/7$ (42.9%) students who accomplished at least 3 experimental puzzles as expected in Bugs, Examples, and Text respectively. This indicates that the majority of students in Bugs group finished the learning activity properly, however, most students in Examples and Text group did not attend to the example code or instructional text.

### Table 3.3 Mean (SD) and median total experimental puzzle completion time in minutes

<table>
<thead>
<tr>
<th></th>
<th>Text (n=7)</th>
<th>Examples (n=6)</th>
<th>Bugs (n=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean time (mins)</td>
<td>29.58 (5.67)</td>
<td>6.08 (1.77)</td>
<td>23.69 (7.37)</td>
</tr>
<tr>
<td>Median time (mins)</td>
<td>28.67</td>
<td>5.98</td>
<td>22.38</td>
</tr>
</tbody>
</table>

Figure 3.1 shows boxplots of completion time for each condition on the first three APs. There is a significant effect of condition on AP completion time ($H(2) = 6.475$, $p < .05$). A post-hoc Mann-Whitney test with Benjamini-Hochberg correction shows a significant difference between Bugs and Examples conditions ($W = 607$, $p < .05$). The results show that the majority of Bugs participants are more time-efficient in solving the first three APs.

Figure 3.2 shows the solution code length for each condition on the three APs. There is no significant effect of condition on the solution code length for solving the AP1 ($H(2) = 2.191$, $p = .334$), AP2 ($H(2) = 1.655$, $p = .437$), and AP3 ($H(2) = 2.197$, $p = .333$). For AP2 and AP3, which focused on loops, all of the participants in the Bugs condition (5/5) used loops to solve the puzzles compared to the Text condition (AP2: 4/7, AP3: 5/7) and Examples condition (AP2 and AP3: 4/6).

Data from this pilot study shows that those who were in the Bugs condition were significantly more time-efficient in solving the first three APs and had non-significantly shorter solution-code lengths than the other conditions. Students in the Bugs condition also performed more consistently, with lower variance in completion time and solution length compared to the other conditions,
indicating that Bugs worked well for all students in the condition. While the Examples condition took significantly less time on the experimental puzzles than the other two conditions, they also had longer AP solution code lengths, compared with the Bugs condition, indicating that they may not have learned the material as well. The non-expected differences in completion times and solution code length between Bugs and the other conditions show promise in our implementation of Bugs as a successful Support.

### 3.4 Discussion

Based on Cognitive Load Theory (CLT) [Swe98] and the results of previous work comparing Supports [McL14], our two hypotheses were that (1) the Examples group would be faster at completing the experimental puzzles and (2) there would be no significant differences in final assessment puzzle (AP) performance between conditions. Hypothesis 1 was confirmed in the significantly lower completion time of the experimental puzzles by the Examples group.

However, Hypothesis 2 cannot be supported by our findings. Differences in the completion times and solution code length between Bugs and the other two Supports on APs suggest that Bugs is the most effective Support we designed for teaching loops. Guided by CLT, both the Examples and Bugs Supports were designed to reduce extraneous load and to increase germane load, which facilitates learning. We hypothesize that both Examples and Bugs Supports successfully lowered extraneous
load by removing problem-solving steps, but the Examples Support also lowered *germane load*, since students were not required to engage with the problem except through simple comprehension questions. This led to decreased learning and worse performance on the APs. While previous studies show erroneous examples are beneficial to those with higher prior knowledge \[ GR07 \], our pilot study also indicates that Bugs may be beneficial to those who had low prior knowledge.

Reflecting on our support designs, the Examples Support had students view the final solution code without seeing the step-by-step construction of the code. This presentation may not have allowed them to as easily reproduce these structures later in the APs. Although the extraneous load imposed on students was low, students received the information passively and could skip the puzzle without understanding the solution. This is evidenced by our data that most students did not answer the questions in the experimental puzzle correctly. Even though we presented conceptual questions at the end of each puzzle, we did not give students an opportunity to apply what they saw in the example immediately. Examples Support in other studies ([SC85; Paa92]) have addressed this by interweaving Examples with problem solving, giving students practice opportunities. Our original design of the Examples Support was based on more recent work, which suggests that Examples with problem solving and Examples alone may be equally effective \[ VG11 \], but our results suggest that interweaving problem solving with Examples might be more effective in the domain of programming.

In the Text condition, students are required to understand the instructions and then solve the puzzle from scratch. Having all instructions presented in the beginning might have increased
cognitive load, especially for puzzles that involved multiple conceptual steps to produce a solution (e.g. recognize the puzzle pattern, create a loop, create loop body). Our data suggest that most students in the Text condition did not apply advanced commands to solve experimental puzzles. Students may have become overwhelmed when they attempted these puzzles (unable to decompose the puzzle into the required steps) and thus couldn't apply the advanced commands correctly. This also might have influenced recall during the APs, as students could not easily map a specific instruction or text from an experimental puzzle to the current one.

In the Bugs condition, each puzzle has an almost-correct solution with one or two errors. As such, like Examples, students’ cognitive load is low compared to Text (students start with a near correct solution shown visibly to them). However, unlike the Examples Support, students cannot skip the puzzle without fixing the errors, forcing them to apply some sort of understanding to solve the puzzle. As such, this could explain the differences in AP performance between these two Supports. The Bugs Support is the only one that has low extraneous load and is likely to induce *germane load* by requiring students to have a solution code that uses the advanced commands. Students have enough mental resource to process the information and figure out the solution. This could explain why not all participants in the Text and Examples conditions used loops in AP2 and AP3 but all in Bugs did.

One main limitation is we do not have a strategy in Text or Examples to scaffold students’ learning of the material when they ignore instructions. In our Examples condition, we asked a conceptual question at the end, but we did not validate the answer. This means students can skip through the puzzle, ignoring the material and never engage in the learning process. Similarly, students in the Text condition can ignore the suggested advanced commands in the instructions and solve the puzzle without them.

In future iterations, we plan to redesign Text to segment the instructions by subgoals rather than all at once in the beginning. We hope this will create better-targeted information for students solving the problem in stages. We further want to rework Examples by interleaving showing Examples with problem solving. Margulieux and Catrambone’s work shows that combining subgoals with Examples and Text may improve students’ learning [MC14]. With further design iterations for Examples and Text, we wish to test the new designs in environments that differ from the pilot one. These include experiments with larger populations and run over multiple sessions. We would also like to apply our Support designs (namely Bugs) into other programming environments like *Snap!* [Gar12] to evaluate its effectiveness in helping students learn programming.

### 3.5 Conclusion

This study provides preliminary insight into the design of instructional supports (Supports) in a programming game. We designed three initial Supports: instructional text (Text), worked examples
(Examples) and buggy code (Bugs), motivating our designs through Cognitive Load Theory and previous implementations. We then looked at these Supports’ performance in an educational setting. We found Bugs may be the most effective for teaching students loops in BOTS, compared with Examples and Text. We were unable to find a difference in effectiveness between Text and Examples. Participants in the Examples condition spent much less time on experimental puzzles compared to the other conditions, but they did not perform as well on assessment puzzles as the Bugs condition. While further work is needed, our initial designs and results provide a foundation for researchers and game developers wishing to add Supports to their programming games.
This chapter was adapted from: Zhi, R. et al. “Exploring the Impact of Worked Examples in a Novice Programming Environment.” Proceedings of the 50th ACM Technical Symposium on Computer Science Education. 2019. [Zhi19a]

The original text has been modified as follows: The Introduction and Background section has been shortened and related work have been integrated into Section 2.3 of Chapter 2, and the section describing iSnap has been removed, since this is covered in Section 2.1.6 of Chapter 2.

This study introduces the design and evaluation of incomplete worked examples in a block-based programming environment called Snap! Both Chapter 3 and this Chapter address RQ1.

Abstract

Research in a variety of domains has shown that viewing worked examples (WEs) can be a more efficient way to learn than solving equivalent problems. We designed a system to display WEs, along
with scaffolded self-explanation prompts, in a block-based, novice programming environment called Snap!. We evaluated our system during a high school summer camp with 22 students. Participants completed three programming problems and had access to WEs on either the first or second problem. We found that access to WEs did not significantly impact students’ learning, but they may have lowered students’ intrinsic cognitive load or change students’ perceived difficulty about following problems. Our results show that WEs save students time initially, compared to writing code, but afterwards these students take longer to complete subsequent parts of the assignment. We find that WEs have the potential to improve students’ learning efficiency when programming, but that these effects are nuanced and merit further study.

4.1 Introduction and Background

In computing classrooms, students are often expected to learn by solving programming problems independently. An alternative approach is to show students worked examples (WEs), that guide the student through solving the problem, step by step. WEs have been studied in a variety of domains [SC85; McL14; Liu16], and the results consistently show that replacing some or all practice problems with WEs results in higher learning efficiency, where students take less time to learn the same amount of content. One way WEs may increase learning efficiency is by reducing students’ cognitive load, or mental effort when learning [Swe06]. Though a number of studies have explored the impact of WEs in the domain of programming [PA85; Mor15; CB07; Vih11; Eri17; VMDC92; TR93], few of these studies have specifically compared WEs to traditional problem solving [TR93], especially in novice programming environments [VMDC92].

In this work, we present the “Peer Code Helper” (PCH), which integrates WEs into the Snap! programming environment [Gar15]. We evaluated the PCH and the impact of programming WEs during a high school summer camp with 22 students. We found that while access to WEs did not significantly impact students’ learning, they may have reduced students’ intrinsic cognitive load or may increase students’ intrinsic cognitive load of a following problem without WEs. Students with WEs on a problem completed more objectives on that problem, but this difference was not significant. Our results suggest that WEs allowed students to work more efficiently on the first few problem objectives, but they took longer to complete consecutive objectives that were not supported with WEs.

4.2 Methods

To investigate the impact of incomplete worked examples (WEs) on novice programmers’ learning outcomes, we designed a “Peer Code Helper” (PCH) feature in Snap! to demonstrate the first few steps of a programming problem as structured WEs. We evaluated the impact of these PCH WEs as
compared with traditional problem solving during a high school coding camp for girls in summer 2018. Specifically, we investigated the following research questions. How does having access to WEs during a programming problem impact: **RQ1**: Students’ learning during the problem? **RQ2**: Students’ perceived difficulty and cognitive load with respect to the problem? **RQ3**: Students’ programming efficiency?

### 4.2.1 The Peer Code Helper

We designed and developed the Peer Code Helper (PCH) interface to present interactive WEs in a block-based programming environment called Snap!, which is used extensively in the Beauty and Joy of Computing (BJC) AP Computer Science Principles curriculum [Gar15]. Our design is based on the implementation of WEs [VMDC92] and self-explanation prompts in other domains [Ber09] as described in the related work. PCH shows students the first few steps for solving the given problem. A step was designed to be a meaningful chunk of code to finish an objective of a problem (e.g. draw a square/circle), though the exact example code may not end up in the final solution. For each step, we first show some starting code (“before code”), without the step completed, and ask the student to run it and answer questions related to the function of the code via scaffolded self explanations, as shown in Figure 4.1. Then we show what the code looks like after the step has been completed (“after code”) and ask them to run the new code and consider why each new piece of code was added. Students can compare the “before” and “after” code and reflect on the changes between the two examples. When showing the next example step, the previous “after” code becomes the new “before” code.

We use the Stencil technique [KP05] to direct students’ attention to the example code by enclosing it within a transparent stencil and hiding other components. The example code is runnable but not editable. When asking questions about the code, we used a *scaffolded* self-explanation prompt to help the student reflect on the code because prior work suggests that scaffolded prompts are more effective than open-ended prompts [Ber09]. Students must fill in the blanks to explain what the “before code” does and then how the new parts improve the “after code”. Students can run the example code and view its visual output. Then students can reflect on the example code and its output, complete the self-explanation prompt, and click the ‘Submit’ button to go to the next step. After finishing all the WE steps, each piece of example code, commented with its explanation, is provided to the student within the Snap! scripting window. Students can then edit the code to build their solution.

### 4.2.1.1 Participants & Procedure

We studied the effects of WEs in Snap! on the first 2 days of a 5-day computing summer camp with 22 female high school students, grades 8-9 (ages 13-15). Five students were excluded from the study
Figure 4.1 Peer Code Helper Interface. Students can run the example code (1), reflect on the output (3), and explain the code (2). Examples are shown side by side. Fill-in-blanks answer choices are marked correct in green or incorrect in red.

(as explained later), leaving 17 students with 6 who identified as African-American, 4 as White, 1 Hispanic, and 6 as multiple ethnicities. About half of the participants (9/17) self-identified as having some programming experience and the rest (8/17) as having none or very little.

We adapted three drawing-themed programming problems with increasing difficulty from the BJC curriculum [Gar15]. The full description of the problems can be found in A.1. The first problem, Daisy Design, is to write a program to draw a pattern of circles. Its common solution has ~7 lines of code using procedures and loops. The second problem, Spiral Polygon, is to draw a polygon in a spiral shape. Its common solution is 7 ~10 lines of code and uses procedures, loops, and variables. The third problem, Brick Wall, uses ~25 lines of code with procedures, loops, variables, and conditionals. Each problem has a set of goal objectives and several stretch objectives in case some students finished early.

The study used a matched pairs, switching replication design as shown in Table 4.1 to investigate the impact of WEs on problem 1, and whether or not it matters which problem receives WE support. In Step 0, the instructor led an introduction to programming in Snap!, including loops, variables, conditionals, and procedures. In Step 1, students took the pre-test and the pre-survey, and were randomly assigned to groups (E1 receiving WEs on problem 1, E2 with WEs on problem 2), with matched pairs according to pre-test scores. The pre-test, and isomorphic post-test1 and post-test2, were adapted from the block-based version of the Commutative Assessments [WW15b], with 16 multiple-choice questions on variables, loops, procedures, and conditionals. Each topic has four types of questions: code tracing, code comprehension, code completion, and WEs (designed to exactly match PCH WEs). Next, the instructor demonstrated PCH (Step 2) and all the students started problem 1 (Step 3). Camp instructors followed a tutoring protocol to ensure that each student received the same quality and quantity of help, using an audio recorder to capture the dialogue of these help episodes, and record who they helped, the help duration (time), and the
type of help (e.g. high-level planning, programming, debugging, interface related). Then students completed the post-test1 and the cognitive load instrument to measure their perceived cognitive load (Step 4). We used the CS Cognitive Load Component Survey (CS CLCS) [Mor14] as our cognitive load instrument. On day 2, students repeated the process for problem 2 (Step 5 ~ 7). Additionally, they completed problem 3 as a performance post-test, without WEs (Step 8) and filled in post-surveys (Step 9).

During the study, we collected data including participants’ interaction with the system (e.g. clicking a button, altering or running their program) and regular snapshots of the students’ code with timestamps when they are modified. During the study, students were also able to ask for help from instructors and volunteers, and we also recorded the frequency and duration of the help requests that students made. After the study, we graded students’ code to measure the code quality and students’ programming performance.

Table 4.1 Study Outline

<table>
<thead>
<tr>
<th>Step</th>
<th>Group E1</th>
<th>Group E2</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Snap/Introduction (taught by camp instructor)</td>
<td></td>
<td>90 minutes</td>
</tr>
<tr>
<td>1</td>
<td>Experience pre-survey + Knowledge pre-test</td>
<td></td>
<td>35 minutes</td>
</tr>
<tr>
<td>2</td>
<td>Introduce the Peer Code Helper</td>
<td></td>
<td>10 minutes</td>
</tr>
<tr>
<td>3</td>
<td>E1: Problem 1 (WEs)</td>
<td>E2: Problem 1 (no WEs)</td>
<td>45 minutes</td>
</tr>
<tr>
<td>4</td>
<td>Post-test1 + Cognitive load survey</td>
<td></td>
<td>25 minutes</td>
</tr>
<tr>
<td></td>
<td>Second Day</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Re-introduce the Peer Code Helper</td>
<td></td>
<td>5 minutes</td>
</tr>
<tr>
<td>6</td>
<td>E1: Problem 2 (no WEs)</td>
<td>E2: Problem 2 (WEs)</td>
<td>45 minutes</td>
</tr>
<tr>
<td>7</td>
<td>Post-test2 + Cognitive load survey</td>
<td></td>
<td>25 minutes</td>
</tr>
<tr>
<td>8</td>
<td>Problem 3 (Brick Wall, no WEs)</td>
<td></td>
<td>45 minutes</td>
</tr>
<tr>
<td>9</td>
<td>Demographics (post-survey) + Cognitive load survey</td>
<td></td>
<td>15 minutes</td>
</tr>
</tbody>
</table>

4.3 Results

Our original data included 22 students, 10 in group E1 (receiving WEs on problem 1), and 12 in group E2 (receiving WEs on problem 2). However, five students were excluded from survey and assessment analysis because they were not present for the whole study (2) or encountered errors in Snap/ that caused them to lose significant programming progress (3). After excluding these students, there are 9 students in Group E1 and 8 students in Group E2.
4.3.1 RQ1: Assessments Analysis

For our analysis of pre- and post-tests, we exclude one additional student from Group E1 who did not fill in post-test1, leaving 8 students in each group. Test results for student’s pre-test, post-test1 and post-test2 are shown in Table 4.2. We compared the pre-test scores between groups and found no statistically significant difference ($t(13.96) = -0.4, p = .64, d = .24$). We then ran a 2x3 mixed ANOVA on students’ test scores, with group (E1 v. E2) as a between-subjects factor and test time (pre-test, post-test1, and post-test2) as within-subjects factor. There was a main effect of test time ($F_{2,28} = 5.26, p < .05, \text{partial } \eta^2 = .27$), indicating students’ test scores changed significantly from pre-test to the two post-tests as shown in table 4.2. We found statistically significant improvements for all students from both pre-test to post-test2 ($t(15) = 3.05, p < .01, d = .30$) and post-test1 to post-test2 ($t(15) = 2.49, p < .05, d = .24$), but not from pre-test to post-test1 ($t(15) = -0.86, p = .40, d = .08$). This suggests that most of students’ learning occurred during the second problem.

<table>
<thead>
<tr>
<th>Group</th>
<th>Pre-test</th>
<th>Post-test1</th>
<th>Post-test2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group E1 (N=8)</td>
<td>8.13 (4.22)</td>
<td>8.38 (3.62)</td>
<td>9.13 (3.83)</td>
</tr>
<tr>
<td>Group E2 (N=8)</td>
<td>7.13 (4.02)</td>
<td>7.50 (3.21)</td>
<td>8.50 (4.21)</td>
</tr>
</tbody>
</table>

There was no main effect of group, indicating that there was not a statistically significant difference in mean test score between the two groups ($F_{1,14} = 0.20, p = .66, \text{partial } \eta^2 = .014$). Additionally, there was no statistically significant interaction between group and test time ($F_{2,28} = 0.13, p = .88, \text{partial } \eta^2 = .009$), indicating that students in both groups had similar learning across the study.

We also analyzed the test score for each type of problem (code tracing, code comprehension, code completion, and WE) but found no statistically significant difference on any of the measurements above. To understand the impact of WEs on help requests, we ran an 2x3 mixed ANOVA on the number of help requests, with group as a between-subjects factor and problem as a within-subject factor. We found no significant main or interaction effects ($F_{2,28} = 2.15, p = .14, \text{partial } \eta^2 = .13$), indicating that WEs likely did not impact the number of help requests students made, and nor did the problem, despite increasing levels of problem difficulty.

4.3.2 RQ2: Cognitive Load Analysis

We used the CS CLCS [Mor14] to measure students’ cognitive load. We excluded data from one additional student from Group E1 for this analysis because that student did not answer the survey after finishing problem 3, leaving 8 students in each group. The average factor scores of cognitive
load in each category are shown in Table 4.3.

Table 4.3 Mean (SD) factor score of cognitive load (IL - Intrinsic Load, EL - Extraneous Load, GL - Germaine Load)

<table>
<thead>
<tr>
<th></th>
<th>Problem 1</th>
<th>Problem 2</th>
<th>Problem 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IL</td>
<td>EL</td>
<td>GL</td>
</tr>
<tr>
<td>Group E1</td>
<td>4.3 (2.4)</td>
<td>3.5 (3.7)</td>
<td>6.6 (3.2)</td>
</tr>
<tr>
<td>N = 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group E2</td>
<td>4.9 (2.6)</td>
<td>3.4 (2.6)</td>
<td>8.6 (1.6)</td>
</tr>
<tr>
<td>N = 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We ran a 2x3 mixed ANOVA on each type of the cognitive load (CL) scores, with group as a between-subject factor and problem as within-subjects factor. We did not find any significant result for both extraneous load and germaine load. For the intrinsic load, there was not a significant main effect of group ($F_{1,14} = 0.10, p = .76$, partial $\eta^2 = .007$) or problem ($F_{2,28} = 1.78, p = .19$, partial $\eta^2 = .11$) factors, but there was a statistically significant interaction between group and problem on the intrinsic load scores ($F_{2,28} = 4.65, p < .05$, partial $\eta^2 = .25$). This indicates that the two groups experienced different levels of intrinsic CL on different problems, which may be due to the WEs. We investigated this further with post hoc t-tests and found that E1 students’ intrinsic load on problem 1 was significantly lower than on problem 2 ($t(7) = −3.51, p < .01, d = 0.83$), but no such results were found for E2. Similarly, we found E2 students’ intrinsic load on problem 2 was significantly lower than on problem 3 ($t(7) = −4.52, p < .01, d = 1.04$), but no such results were found for E1. This suggests that both groups experienced lower intrinsic CL on problems with WEs than on the following, more difficult problem without WEs. Comparing the two groups, we find that the difference in intrinsic load between E1 and E2 is not statistically significant for problem 1 ($t(13.91) = 0.40, p = .69, d = 0.20$). However, it is significant for problem 2, with a large effect size ($t(13.19) = −2.33, p < .05, d = −1.16$). One interpretation of these results is that students experienced reduced intrinsic CL when using WEs. This would be consistent with results from work by Gerjets et al., who evaluated “modular worked examples,” which, like our WEs, emphasized breaking a problem down into steps [Ger06]. However, another interpretation is that students experienced higher intrinsic CL on problem solving that followed a WE, which made it seem harder by contrast.

4.3.3 RQ3: Programming Efficiency

Previous research in other domains has shown that replacing problem solving exercises with worked examples (WEs) can increase learning efficiency by decreasing the time that students take to complete the exercises without affecting how much they learn. Similarly, RQ3 investigates the impact of
WEs on the time taken to complete programming practice problems, or programming efficiency. However, because most students did not complete the whole problem within the allotted 45 minutes, we analyze completion time for smaller objectives within each problem.

To investigate students’ programming efficiency, two researchers graded all students’ submitted code by marking each problem’s objectives as either complete or incomplete. For complete objectives, the researchers also marked the time when the student first completed the objective. The three problems (Daisy Design, Spiral Polygon and Brick Wall) had 9, 5 and 5 objectives, respectively. The two researchers reached an inter-rater reliability between 0.744 and 0.979 for first 15 student submissions (5 in each problem). Then the researchers discussed and resolved all the conflicts and graded the rest of student code separately. On the first problem, 2 out of 9 objectives were covered in the WEs (22%) and on the second problem, 1 out of 5 was (20%); we refer to these as “WE objectives.” On each problem we removed some students from our analysis who lost their work due to errors in Snap!, leaving 9, 10, and 10 students in E1 and 10, 9, and 10 students in E2 on the three problems, respectively.

Figure 4.2 shows the average number of objectives completed for each group, over time. On both Daisy Design and Spiral Polygon, the group with WEs (E1 and E2, respectively), had more objectives completed at all times throughout the 45 minute work period, while on Brick Wall (with no WEs) there is little difference. These differences between groups is most apparent at the beginning of the curves, when students are completing the WE objectives, though the same trends hold, even if we do not include the WE objectives. On Daisy Design, it took the WE group (E1) a median 3.5 minutes to complete the WE portion of the problem. It took the non-WE group (E2) a median 22.5 minutes to accomplish the same WE objectives by writing code, with 20% of students never finishing the WE objectives in the 45 minutes given. A Mann-Whitney U Test shows that, even ignoring students who did not finish the objectives, this difference in time is significant ($W = 2$; $p = 0.001$). Similarly, on
Spiral Polygon, the WE group (E2) took a median 2.5 minutes on the WE portion of the problem, and the non-WE group (E1) took 30.4 minutes, with 40% of students failing to achieve the objectives on 45 minutes. This difference in time is also significant ($W = 54; p < 0.001$). It is not surprising that the WE group, who only had to read code and answer questions, finished this content much faster (6-12 times faster), but the magnitude of the difference is worth noting. Based on these results, we can say the WEs saved students lots of time (19-24 minutes on a 45 minute problem), and allowed all students to finish the WE objectives (compared to only 80% and 60% for non-WE students).

However, looking past the WE objectives in Figure 4.2, we see that the differences between the two groups closes over time. On Daisy Design, the WE group (E1) end with a median 6 objectives complete (67%), compared to 5 objectives (56%) for the non-WE group (E2), and the difference is not significant ($W = 56.0; p = 0.382$). On Spiral Polygon, the WE group (E2) ends with a median 2 objectives complete (40%), compared to 1 objective (20%) for the non-WE group (E1), and the difference is not significant ($W = 31.5; p = 0.270$). This suggests that WEs may have impacted students' overall performance, but the difference was not significant. One possible explanation for the lack of a significant difference in programming is that while WEs save students time initially, it takes those students additional time to process the WE code they are given, and they work less efficiently at writing code after receiving a WE.

We investigated how the programming efficiency of students who just received WE code compares to those who just wrote equivalent code on their own. Starting from the time each student completed the WE objectives, we calculated how much additional time it took them to complete the next 1, 2 and 3 objectives. Some students finished the WE objectives slowly, or not at all, and ran out of time before they could complete additional objectives. However, for most objectives, enough students finished that we can still calculate the median time for each group (when 50% of students finished and objective) or the first-quartile (Q1) time (when 25% of students finished). To do this, we make the simplifying assumption that the students who completed the WE objectives the slowest would have completed additional objectives no quicker than the other students in their group.

The results of this analysis are given in Table 4.4. For Daisy Design, we see that the median student in the WE-group (E1) took 9.5 minutes to complete their next objective, while the median student in the non-WE group (E2) took 0 minutes (they accomplished the next objective at the same time as the WE objectives). This occurred because the next objective was to move code into a procedure, and many students in the non-WE group simply wrote their code in a procedure from the beginning. The next two objectives also take the WE group longer. For Spiral Polygon, no additional objectives were completed by more than half of students, so we can only compute the Q1 time. Similarly, it takes the WE group much longer to accomplish one additional objective (Q1 = 12.0m) compared to the non-WE group (Q1 = 2.6m), but this gap quickly reduces for the second objective, and even reverses by the third (where not even 25% of non-WE of students had completed the third objective by 39 minutes). From this, we conclude that, after completing the WE-supported objectives, students with
Table 4.4 The first-quartile (Q1) and median (Med) time (in minutes) taken by the WE and non-WE groups to complete 1, 2 and 3 additional objectives after the WE objectives, as well as the proportion that Comp. each objective

<table>
<thead>
<tr>
<th>Add'l Objs</th>
<th>WE Group</th>
<th>Non-WE Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q1 time</td>
<td>Med time</td>
</tr>
<tr>
<td>Daisy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE+1</td>
<td>7.5</td>
<td>9.5</td>
</tr>
<tr>
<td>WE+2</td>
<td>14.2</td>
<td>24.8</td>
</tr>
<tr>
<td>WE+3</td>
<td>23.3</td>
<td>44%</td>
</tr>
<tr>
<td>Polygon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE+1</td>
<td>12.0</td>
<td>44%</td>
</tr>
<tr>
<td>WE+2</td>
<td>13.0</td>
<td>44%</td>
</tr>
<tr>
<td>WE+3</td>
<td>29.4</td>
<td>44%</td>
</tr>
</tbody>
</table>

WEs initially worked less efficiently, especially since non-WE students could accomplish additional objectives while completing the WE objectives (e.g. putting code into a procedure). However, it also seems that this difference decreases over time. In addition, the difference never exceeded 15 minutes on Daisy Design, and 10 minutes on Spiral Polygon, which is still far less time than the WEs saved the median student on each problem (19 and 24 minutes, respectively). This helps to explain why the WE group accomplished more objectives on both problems, but the difference was not significant.

4.3.4 Post-survey Feedback

We asked students for feedback on the Peer Code Helper (PCH) at the end of the study. When asked if they would like to use the PCH on a future problem, 35% of students said yes and 12% said no, with the rest uncertain. We found that the 2 students said “no” had very high pre-test scores (over 75%), indicating that some more advanced students may not appreciate WEs, as found in previous work [Kal03]. This may also suggested by the expertise reversal effect (described in Section 2.2.1 of Chapter 2), that experienced learners may benefit less from instructional support. The participants who were uncertain about the PCH indicated that they prefered the challenge of working independently, saying, “it's good to have a challenge, but it's also nice... to make it a little bit easier.”, and “I liked the ... feeling of success from figuring it out for myself, but sometimes the instructions are really helpful when I get stuck.” Students’ desire for independence is a common reason that they avoid help when programming, and overcoming this barrier can be very challenging [Pri17b]. However, many students wrote about how they appreciated the PCH, emphasizing how it helped them to “see how to go from one step to the next" and that the PCH “gave me somewhere to start coding from.” When asked about their favorite parts of the PCH, students specifically mentioned how it gave “comments beforehand to help us understand how to begin the code.” Paired with the other results presented here, this indicates that the PCH is an overall improvement to Snap!, which students
generally prefer over purely independent programming.

### 4.4 Discussion

We did not find significant differences in learning based on post-test scores between groups. However, the results suggest that most of students' learning occurred during problem 2. This may be because students solve problem 2 on second day of the study, after having more time to reflect and digest the concepts learned in problem 1. There was no difference in learning between groups on the WE problems. This is consistent with previous WE studies [McL14; Sal10], where students learned more efficiently with WEs, but they did not learn more, since both groups were exposed to the same learning content. The problems in our study contained core objectives (e.g. draw the daisy design), followed by more open-ended objectives (make the design colorful) which do not include new programming concepts. Since most students got through the core objectives in both groups, it is not surprising that there was no significant difference in their learning.

We found significant differences between the groups' intrinsic cognitive load for problem 2 but not for problem 1. It is interesting that we observed differences in intrinsic load, which is assumed to be inherent in the learning task, rather than extraneous load, which is assumed to be the result of our instructional design. This may be because WEs represent an inherently different learning task than problem solving. We also found both groups experienced higher intrinsic load on problems without WEs that followed problems with WEs. This indicates that our design of WEs may increase students' perceived difficulty of a following problem without WEs.

Our analysis of programming efficiency suggests that initially, WEs save students considerable time in completing programming objectives (up to half of the time allotted), but that students take longer to complete later objectives. We hypothesize this is because students require time to process the WE code and begin to write their own code. However, the time saved is longer than the time students need to complete additional objectives, which allowed students who received WEs to complete more objectives overall on both problems 1 and 2, although this difference was not significant due to the low sample size.

**Threats to Validity:** This is an exploratory study with a small sample size and important limitations. All of the programming problems used in this study involved drawing patterns, and our design of WEs may not generalize to other types of problems (without visual output). The participants self-selected to join the summer camp and may have been more motivated. However, summer campers may have taken the assessments less seriously than other populations. Our small sample size makes it difficult to rule out possible differences between groups, since we may not have the statistical power to detect these differences. Participants learning gains may also have come from repeated exposure of tests.
4.5 Conclusion

This study evaluates the impact of incomplete WEs, along with scaffolded self-explanation prompts, on learning, cognitive load, and efficiency in block-based programming activities in a high school summer camp. We found that WEs have the potential to reduce students' intrinsic cognitive load. It is also possible that WEs make the following problem seems harder. Our results also suggest that programming WEs may improve students' learning efficiency but that students do require additional time to process WEs before they can construct their own code.

The two studies in Chapter 3 and this Chapter showed that worked examples have the potential to improve students learning efficiency, which is consistent with studies about worked examples in other domains [McL14]. However, the effect of worked examples is affected by the design of worked examples. In Study 1, we did not pair practice problems after presenting worked examples, and participants seem to skip the puzzles instead of understanding the example solution. While in the erroneous worked example condition, participants need to locate the error and fix the buggy code. This design let participants put mental effort to learn the programming concepts inherent in the puzzles. Similarly, in Study 2, with the incomplete worked example design, students can learn from the examples and accomplish the remaining objectives based on the example code. The results from both studies motivate my work to design new types of instructional support, that strike a balance between problem solving and worked examples, as described in Chapter 5 and Chapter 6.
This chapter was adapted from: Zhi, R. et al. (in press) “Evaluating the Effectiveness of Parsons Problems for Block-based Programming.” Proceedings of the International Computing Education Research Conference. 2019. [Zhi19b]

The original text has been modified as follows: The Related Work Section has been removed and integrated into Section 2.3 of Chapter 2, and the section describing iSnap has been removed, since this is covered in Section 2.1.6 of Chapter 2.

This study introduces the design and evaluation of Parsons problems in Snap! and RQ2 is addressed in this Chapter.

**Abstract**

Parsons problems are program puzzles, where students piece together code fragments to construct a program. Similar to block-based programming environments, Parsons problems eliminate the need to learn syntax. Parsons problems have been shown to improve learning efficiency when compared to writing code or fixing incorrect code in lab studies, or as part of a larger curriculum. In this study, we directly compared Parsons problems with block-based programming assignments in classroom settings. We hypothesized that Parsons problems would improve students’ programming efficiency
on the lab assignments where they were used, without impacting performance on the subsequent, related homework or the later programming project. Our results confirmed our hypothesis, showing that on average Parsons problems took students about half as much time to complete compared to equivalent programming problems. At the same time, we found no evidence to suggest that students performed worse on subsequent assignments, as measured by performance and time on task. The results indicate that the effectiveness of Parsons problems is not simply based on helping students avoid syntax errors. We believe this is because Parsons problems dramatically reduce the programming solution space, letting students focus on solving the problem rather than having to solve the combined problem of devising a solution, searching for needed components, and composing them together.

5.1 Introduction

Parsons problems [PH06] are code-completion problems, which require students to rearrange mixed up blocks of code to create the correct solution to a programming problem (see Figure 5.1). Parsons problems have been shown to be more efficient in helping novices learn to program than writing equivalent code, fixing buggy code [Eri17; Eri18], or reading an instruction-based tutorial [Har15]. In these studies, students can solve Parsons problems more quickly than other forms of practice, while performing just as well or better on post test problems. However, these promising results come from short, lab-based controlled studies, with voluntarily recruited populations, and procedures that differ from many classroom contexts. For example, in a 2.5 hours study, Ericson et al. compared worked examples paired with Parsons problems to worked examples paired with problem solving practice [Eri17; Eri18] and measured learning gains for the whole instructional material. More work is needed to understand how this effect might generalize across a semester, and whether Parsons problems are still more effective without paired worked examples.

It is also important to understand what features of Parsons problems make them effective. There are many ways that solving a Parsons problem differs from writing equivalent code. In Parsons problems, students drag and drop code, rather than typing it, they construct a solution from a limited set of available code blocks, and these code blocks may contain multiple lines of code. What role do these elements play in Parsons’ problems effectiveness? For example, in block-based programming environments, students also construct code with a limited set of drag-and-drop blocks, rather than by typing, which has been shown to improve programming efficiency [PB15] and learning [WW17]. It is therefore worth investigating whether the previously observed benefits of Parsons problems also apply in a block-based programming environment, which already offers similar benefits to Parsons problems.

In this study, we compared Parsons problems with traditional problem solving (writing code) during 6 weeks of a non-majors CS0 course. We investigated whether our Parsons problems in-
creased students’ programming efficiency. We also explored how Parsons problems save students time, by measuring differences in programming behavior between students who solved Parsons problems and those who wrote equivalent code. We used a quasi-experimental design, and compared student work across six semesters. We hypothesized that (1) students who solve block-based Parsons problems will complete the learning activities faster, and (2) perform no worse than students writing block-based code. We confirmed both of our hypotheses, finding that Parsons problems saved students 15-17 minutes on each lab assignment, between 43.6% and 65.2% of the time they would have spent writing code, and found no evidence that they negatively impacted students’ later performance. Our analysis of log data suggests that this time was primarily saved by limiting students’ interactions to only those blocks that are part of the solution. The primary contributions of this work are: 1) evidence that Parsons problems improve learning efficiency compared to writing block-based code in a classroom context, 2) an investigation into how Parsons problems contribute to programming efficiency, and 3) the design of Parsons problems for block-based programming.

5.2 Method

To investigate the impact of Parsons problems on novice programmers’ problem solving performance, we designed and integrated Parsons problems into Snap!. We used a quasi-experimental design, comparing student performance in a non-majors CS0 course on 6 programming assignments and a project across 6 semesters. In the Spring 2019 semester, 3 of the 6 assignments (the labs) were presented as Parsons problems, rather than the traditional problem solving used in prior semesters. We compare student solution time and code quality and their problem-solving behaviors while solving the programming problems to investigate the following research questions: How do block-based Parsons problems compare to writing code on: RQ1: students’ programming efficiency on the problem? RQ2: students’ performance on post-test problems and final project? RQ3: students’ hint usage while working on the post-test problems? RQ4: How do Parsons problems impact students’ problem solving behavior on the problem and post-test problems? We hypothesized that students who do Parsons problems instead of writing code during programming labs will: H1: Accomplish the lab assignment faster than previous semesters’ students. H2: Perform no worse than previous semesters’ students on the following ‘post-test’ homeworks (measured by grades, time on task, and hint usage). H3: Perform no worse than previous semesters’ students on the final project.

5.2.1 Parsons Problems Design

We designed and implemented block-based Parsons Problems in iSnap (Figure 5.1). For each problem, we separated an expert solution into individual blocks, which we present in the code palette (Figure 5.1(1)). Students can drag blocks together or drag variables into block inputs, but cannot
duplicate or delete blocks. In traditional Parsons Problems, students manipulate entire lines of code, but in our implementation, students manipulate individual blocks, just like in Snap!, and it may take multiple blocks to complete a line (e.g., the “turn” statement in Figure 5.1). We specifically designed our Parsons problems in this way, so that students would not be disadvantaged when doing later assignments as they have practiced with how blocks interact with one another. In Snap!, when a student defines a procedure (a “custom block”), they use its parameters by dragging a copy of the parameter variable from the block definition header (see Figure 5.1). For Parsons problems, we already know how many times these parameter variables are needed in the expert code. Therefore, students can only use the parameter variables the predetermined number of times, and when they run out, the input variable is greyed out, as shown in Figure 5.1(D). This design guarantees that the solution will not contain extraneous variables and helps students focus on the relationship between blocks, variables, and parameters. As suggested by the element interactivity hypothesis of Cognitive Load Theory [Swe10b], reducing the number of learning elements (which often correspond to UI elements) that students interact with should decrease the cognitive load imposed on students.

Our Snap! Parsons problems have three modes for inputs: editable, editable with a default value, and fixed. Editable inputs have a white background, as in Snap!, and allow students to drop in variables. We added a default value to some editable inputs, shown in faded gray, as in Figure 5.1(A)), so that students can experiment with the blocks before they are done. Fixed inputs cannot be changed, and are entirely greyed out, as shown in Figure 5.1(B). We designed this feature with the intent to help students gain the knowledge for using variables, since we have observed in our previous study that novices have trouble using variables in Snap! [Zhi18b].

Students can run the code whenever they want to test it. They can click the ‘Reset Sprite & Clear Stage’ button to clean the stage area and put the sprite in its default position (Figure 5.1(4)). Students can also click the undo button to remove changes one step at a time (Figure 5.1(3)). We also hide and disable elements such as the stage icon or functions that are not related to the problem. This design eliminates extraneous elements in the environment, taking the “weeding” suggestion from Mayer and Moreno to reduce cognitive load in multimedia learning environment [MM03].

After students run their code a few times, we enable ‘Check My Work’ help button (Figure 5.1(2)). Unlike MobileParsons [Kar12], which deactivated feedback for a short period (e.g. 15s) to prevent help abuse, our system does not limit help requests. This is to keep our design consistent with the hint system used for all the class Snap! assignments. For assignments with multiple sprites, we support the “Check My Work” feature for each sprite in a program, independently. If the current sprite’s program is correct, the system would encourage them to test other sprites until all are correct, and then it will suggest submitting the assignment. Otherwise, the system would highlight any misplaced blocks in the programming area, as shown in Figure 5.2. Our Parsons problems may contain multiple solutions, and our system can highlight misplaced blocks based on placements that no known solutions would accept. For example, ‘pen down’ should be put before ‘pen up’ and
the ‘move steps’ blocks, but it does not matter whether it is right after the hat block or the ‘set pen size’ block. After they finish, students can upload solutions to the server and download a copy for reference.

![Figure 5.1](image1.png)

**Figure 5.1** Parsons problem interface in Snap!. Available code blocks are shown in the code palette.

![Figure 5.2](image2.png)

**Figure 5.2** A hint example: yellow highlights misplaced blocks or variables, blue shows input that needs a variable.

### 5.2.2 Participants

In this study, the participants were undergraduate students, with minimal prior programming experience, enrolled in a non-majors CS0 course at a U.S. research university over 6 semesters (Fall 2016 through Spring 2019). Students are typically non-STEM majors, but we did not collect student demographic information as part of this study. This course introduces basic programming concepts in Snap! including variables, loops, conditionals, procedures, and lists. All semesters of the course
were taught by the same instructor, and the weekly programming labs were led by undergraduate TAs in 3-4 sections. Some or all of the TAs changed each semester. There were 46-68 students each semester (M=50; see Table 5.2). In this course, students learned Snap! through 3 lab assignments, 3 homeworks, and a final project (described in the next section). During lab sessions (110 minutes), TAs introduced relevant Snap! concepts, administered a quiz, and provided guidance while students worked on the labs. After completing the lab, students can leave early or start the week's homework assignment. In our data, few students chose to start their homework (including in the Spring 2019 semester).

5.2.3 Study Design

Our study employed a quasi-experimental design. Each semester, students completed 3 programming labs and 3 homeworks, as explained in detail below. Students in the Spring 2019 semester served as the experimental group, and the 3 lab assignments (but not the 3 homeworks) were given as Parsons problems, rather than traditional problem solving. We refer to Spring 2019 as X19 to clearly indicate that this was our experimental semester. Students from the Fall 2016 (F16), Spring 2017 (S17), Fall 2017 (F17), Spring 2018 (S18), and Fall 2018 (F18) semesters served as control groups, as they did not have access to Parsons problems. For all semesters, the 3 homeworks were given as problem solving and served as assessment problems. Additionally, all students in the control and experimental groups could get help from TAs during labs and use iSnap's on-demand hints for homeworks. We did not use a randomized, controlled experiment because it would not be fair or feasible to give different assignments to students in the same class for a given semester. Except for introducing the help features and the Parsons problems right before the first lab in each semester, we were not involved in teaching the course or labs. As mentioned in section 5.2.1, our Parsons problems require students to compose a correct solution to a given problem from a minimal set of needed blocks and variables. Before the first lab in X19, the first author used a demo project to introduce the Parsons problems and help features to students. Each lab is followed by a homework assignment, used as an immediate post-test. The final project also served as a summative, delayed post-test.

Assignments & Course Project. Students completed 3 labs assignments: PolygonMaker, Pong1, and GuessingGame1 (GG1); and 3 homeworks: Squiral, Pong2, and GuessingGame2 (GG2). Students do the lab assignments in person during the lab time, and then do a similar, follow-up homework assignment outside of class. The first lab, PolygonMaker, is to write a program to draw any polygon. Its common solution is 5 lines of code using procedures (functions), loops, and variables. The following homework, Squiral, requires students to write a program to draw a spiraling square-like shape using loops, variables, arithmetic operators, and procedures. Common solutions for Squiral contain 7-10 lines of code. The second lab, Pong1, simulates the classic arcade game Pong where
one player can control a paddle to hit a ball back and forth to earn points. The Pong1 lab requires
students to program the ball and paddle behaviors, with common solutions using ~18 lines of code
using loops, variables, conditionals, and user interactions. The homework Pong2 extends Pong1 to
allow two players to compete with each other and its common solution contains ~29 lines of code
using similar concepts. The third and last lab assignment, GG1, requires students to program a game
that asks players to guess a number generated by the computer until they guess correctly. It informs
players if they guessed too high or too low. Common solutions for GG1 have 12-18 lines of code using
loops, conditionals, variables, and arithmetic operators. The GG2 homework requires students to
extend GG1 by selecting a random number from player-set range and using extra variables and lists
to record the total number of guesses made and the value of each guess. Common solutions for
GG2 have 25-30 lines of code using loops, conditionals, variables, arithmetic operators, and lists.
After all labs and homeworks, students complete a Snap! project that requires students to build a
user interface prototype for a product using loops, conditionals, procedures, variables, and user
interaction.

Because the course curriculum has changed somewhat over 3 years, each assignment can only
be compared across a subset of the semesters. Table 5.1 shows a list of all the assignments and
semesters. For a given assignment, we excluded any semester from comparison if the assignment
was not offered in that semester, if its instructions did not match those used in X19, if there was
another experiment being run during that semester, or if there were unique circumstances (e.g.
canceled classes/labs). These cells in the table have dashes to show that semester was excluded.
Specifically, Pong1 and Pong2 were not offered until S18, GG2 was modified in F18, PolygonMaker
and Squirrel were modified for just the F18 semester, there was an experiment during Squirrel for S18,
and there was a class cancellation during PolygonMaker for S18. In addition, due to a power outage,
we do not have log data for Pong2 in F18, so we can only compare grades but not time or hint usage.

5.3 Results

To answer our research questions, we compared the X19 students’ homework performance with
previous semesters’ students’ (the control) based on the code quality and time for finishing the
assignment, evaluated their performance on the final project, and also analyzed students’ program-
ming traces to identify their problem solving behavior patterns.

5.3.1 Time on Task

To answer RQ1, we investigated the time that students took to finish the assignments. We did not
investigate the time that students spent working on the final project because the project is related to
their own majors and does not have specific goals like other assignments. Specifically, we calculated
the time that students spent working on the problem instead of the time between opening and submitting the assignment. Since students may just open the assignment and work on other things instead, we exclude the idle time, defined as any period of 5 minutes or more without interactions in iSnap. Table 5.1 shows the average time along with standard deviation that students took for finishing each assignment in each semester. The results show that students spend much less time on Parsons problems than traditional problem solving. The Parsons problems save students nearly 17 minutes on average in PolygonMaker (65.2% of the total problem solving time), 15 minutes (43.6%) on average in Pong1, and 17 minutes (60.6%) on average in GG1. For each assignment, we performed a Kruskal-Wallis test\(^1\) to determine if these differences in active time across semesters were significant. As shown in the final column of Table 5.1, the differences were significant for each lab assignment, but not for any of the homework assignments. For each lab assignment, we performed a post hoc Dunn's test, using the Benjamini Hochberg procedure to control the false discovery rate at 5% [BH95; Dun64], to identify pairwise differences in completion time between semesters. In each case, X19 had significantly lower lab completion time than all other semesters (all \(p < 0.001\)). These findings support H1, that students with Parsons problems will complete these lab assignments more quickly. The findings also partially support H2, since there were no significant differences across semesters in the time taken to complete homework assignments. For most assignments, there were no significant differences in completion time among the control semesters (without Parsons problems). However, for Pong1, we also found that F18 < S18 (\(z = 5.78, p < 0.001\)), and for GG1, we found that F18 < F16 (\(z = 3.35, p = .002\)), F18 < F17 (\(z = 2.90, p = .007\)), F18 < S18 (\(z = 2.15, p = .048\)), S18 < F16 (\(z = 3.17, p = .003\)) and S18 < F17 (\(z = 2.75, p = .010\)). We would discuss the potential reasons for this result in section 5.4.

### 5.3.2 Effectiveness

To answer our second research question, we compared students' grades for each homework assignment across semesters, as reported in Table 5.2. Students who did not complete the assignment received a grade of 0, and we found the dropout rate is consistent across semesters, ranged from 0% to 10.9%. For completeness, we also report grades for the lab assignments, but we note that this comparison is less meaningful, since students can get help from TAs and have historically performed uniformly well on these assignments. We performed a Kruskal-Wallis test to determine if these differences in grades across semesters were significant. As shown in Table 5.2, we found significant differences for 3 assignments: PolygonMaker (in-lab), Pong2 (HW) and GG2 (HW). For these assignments, we performed post hoc Dunn's tests, using the Benjamini Hochberg procedure to control the false discovery rate at 5%, to identify pairwise differences in grades between semesters. For PolygonMaker, we found that F16 had a significantly higher score than all other semesters (F16

---

\(^1\)Nonparametric tests were used because the distribution of the data was non-normal, indicated by the Shapiro-Wilk test.
Table 5.1 Average time (with Standard Deviation) for accomplishing the assignment in different semesters.

<table>
<thead>
<tr>
<th></th>
<th>F16</th>
<th>S17</th>
<th>F17</th>
<th>S18</th>
<th>F18</th>
<th>X19</th>
<th>Kruskal-Wallis Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolygonMaker (Lab)</td>
<td>23.6 (11.3)</td>
<td>27.5 (10.7)</td>
<td>30.0 (15.8)</td>
<td>-</td>
<td>9.3 (4.1)</td>
<td>-</td>
<td>$\chi^2(3) = 82.4, p &lt; .001$</td>
</tr>
<tr>
<td>Pong1 (Lab)</td>
<td>-</td>
<td>-</td>
<td>41.6 (11.0)</td>
<td>25.8 (17.2)</td>
<td>18.8 (6.9)</td>
<td>-</td>
<td>$\chi^2(2) = 72.8, p &lt; .001$</td>
</tr>
<tr>
<td>GG1 (Lab)</td>
<td>31.3 (12.1)</td>
<td>28.4 (10.5)</td>
<td>30.6 (11.8)</td>
<td>24.1 (10.9)</td>
<td>23.3 (8.5)</td>
<td>11.0 (6.2)</td>
<td>$\chi^2(5) = 111.4, p &lt; .001$</td>
</tr>
<tr>
<td>Squirrel (HW)</td>
<td>23.6 (21.5)</td>
<td>20.4 (15.3)</td>
<td>28.5 (31.8)</td>
<td>-</td>
<td>19.0 (14.9)</td>
<td>-</td>
<td>$\chi^2(3) = 4.4, p = .23$</td>
</tr>
<tr>
<td>Pong2 (HW)</td>
<td>-</td>
<td>-</td>
<td>47.6 (35.8)</td>
<td>-</td>
<td>51.7 (31.0)</td>
<td>-</td>
<td>$\chi^2(1) = 1.5, p = .22$</td>
</tr>
<tr>
<td>GG2 (HW)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>62.3 (50.1)</td>
<td>50.8 (24.0)</td>
<td>$\chi^2(1) &lt; 0.1, p = .99$</td>
</tr>
</tbody>
</table>

vs S17, $z = 4.72, p < 0.001$; F16 vs F17, $z = 3.40, p = .002$; F16 vs X19, $z = 2.20, p = .04$), and that X19 had a higher score than S17 ($z = 2.30, p = .04$). For Pong 2, we found that S18 had a significantly lower score than all other semesters (S18 vs F18, $z = 2.27, p = .04$; S18 vs X19, $z = 2.95, p = .0095$). For GG2, we found the X19 had a significantly higher score than F18 ($p = .006$). We would talk about the possible explanations in section 5.4.

Our data show no evidence that students with Parsons problems on lab assignments perform any worse than those with traditional problem solving, supporting H2. While this lack of evidence is not sufficient to claim that the two groups performed equally well in a statistical sense, the mean X19 grade was never more than 1 percent worse than any other semester and was significantly higher than some semesters for Pong2 and GG2.

5.3.3 Log Analysis

In this section we discuss results for RQ3 and RQ4 to investigate the impact of Parson’s problems on later programming behaviors. **RQ3:** How do block-based Parsons problems compared to writing code on students’ hint usage while working on the post-test problems? **RQ4:** How do Parsons problems impact students’ problem solving behavior on the lab problem and on post-test problems?

5.3.3.1 Hint Usage

We investigated the number of times that students asked for hints across semesters and the result is shown in Table 5.3. We performed a Kruskal-Wallis test and did not find significant differences on the three homeworks between X19 and other semesters.
Table 5.2 Average class grades (with Standard Deviation) for each assignment in different semesters.

<table>
<thead>
<tr>
<th></th>
<th>F16 (68)</th>
<th>S17 (50)</th>
<th>S18 (46)</th>
<th>S18 (49)</th>
<th>X19 (48)</th>
<th>Kruskal-Wallis Test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PolygonMaker</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Lab)</td>
<td>64.6 (16.7)</td>
<td>60.9 (13.7)</td>
<td>63.1 (13.3)</td>
<td>-</td>
<td>-</td>
<td>$\chi^2(3) = 24.7, p &lt; .001$</td>
</tr>
<tr>
<td><strong>Pong1 (Lab)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>60.6 (20.2)</td>
<td>67.3 (9.9)</td>
<td>66.0 (11.1)</td>
</tr>
<tr>
<td><strong>GG1 (Lab)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>65.8 (14.6)</td>
<td>65.7 (14.4)</td>
<td>66.1 (13.9)</td>
<td>61.1 (21.6)</td>
<td>68.1 (10.1)</td>
<td>67.0 (14.1)</td>
</tr>
<tr>
<td><strong>Squirrel (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>81.3 (23.6)</td>
<td>81.0 (31.7)</td>
<td>81.9 (29.8)</td>
<td>-</td>
<td>-</td>
<td>81.5 (24.4)</td>
</tr>
<tr>
<td><strong>Pong2 (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>74.5 (31.4)</td>
<td>87.5 (22.1)</td>
<td>86.2 (27.5)</td>
</tr>
<tr>
<td><strong>GG2 (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td>77.1 (26.5)</td>
</tr>
<tr>
<td><strong>Project</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25.1 (7.4)</td>
<td>26.0 (4.9)</td>
<td>27.0 (4.8)</td>
</tr>
</tbody>
</table>

Table 5.3 Average hint requests (with Standard Deviation) of students for homeworks. The percentage of students who request hints are shown with %.

<table>
<thead>
<tr>
<th></th>
<th>F16 (68)</th>
<th>S17 (50)</th>
<th>S17 (51)</th>
<th>S18 (46)</th>
<th>S18 (49)</th>
<th>X19 (48)</th>
<th>Kruskal-Wallis Test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Squirrel (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.23 (6.9)</td>
<td>3.65 (8.0)</td>
<td>1.40 (5.6)</td>
<td>-</td>
<td>-</td>
<td>5.0 (9.08)</td>
<td>$\chi^2(3) = 5.83, p = .12$</td>
</tr>
<tr>
<td><strong>Pong2 (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10.28 (24.7)</td>
<td>-</td>
<td>17.0 (30.3)</td>
<td>$\chi^2(3) = 3.65, p = .06$</td>
</tr>
<tr>
<td><strong>GG2 (HW)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>21.80 (42.0)</td>
<td>14.62 (26.2)</td>
<td>$\chi^2(3) = 0.63, p = .43$</td>
</tr>
</tbody>
</table>
5.3.3.2 Problem solving behavior

To figure out why Parsons problems save students time, we analyzed students’ code traces to investigate their programming patterns. We found several common patterns that differentiate Parsons problems with writing block-based code by manually examining students’ code traces. These patterns demonstrate why solving Parsons problems is efficient. We hypothesize that the block-based Parsons problems can save students time by reducing the time spent on, or even avoiding, unproductive patterns. We investigated three potential unproductive behaviors: searching for blocks to use, editing block inputs, and testing irrelevant blocks. These behaviors were selected based on a manual inspection of lab programming traces from control students who took much longer than the median time, looking for patterns that did not seem to result in significant differences in the student code, but took longer than a few seconds. Once these candidate behaviors were determined, we compiled statistics for all students for these behaviors on each homework and control group lab. X19 students could not perform these behaviors within Parsons problems. We hypothesized that although Parsons problems prevent these unproductive behaviors during the lab, they would not disproportionately increase these unproductive times during the homework.

Search blocks. To write code in Snap!, students have to search and find the blocks they need from the code palette. They need to change categories and skim the code palette to find desired blocks. Prior work investigating block-based programming examples show that novices have trouble finding blocks presented in examples [Ich17]. We argue that Parsons problems organize all the needed blocks together and hence reduce search time. To investigate this for students in the control group, we calculated the time spent searching for blocks as any time between a category switch and the addition of a block into the script window. We found that students in the control group spent 1.32 (SD = 0.99, 4.98% of the total problem solving time), 2.66 (SD = 1.33, 8.0%), 2.66 (SD = 1.47, 9.5%) minutes on average for finding blocks in PolygonMaker, Pong1, and GG1 respectively. A Spearman’s rank-order correlation was run to assess the relationship between the time spent finding blocks and solving the lab assignments. There was a statistically significant, strong positive correlation between the proportion of time spent finding blocks and solving PolygonMaker (ρ = 0.75, p < 0.001), Pong1 (ρ = 0.59, p < 0.001), and GG1 (ρ = 0.69, p < 0.001). For the homework assignments, we found students in the control group spent 1.47 (SD = 1.37, 6.1%), 2.46 (SD = 1.95, 5.2%), 3.81 (SD = 4.37, 6.1%) minutes on average for finding blocks in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.09 (SD = 1.05, 5.8%), 2.74 (SD = 2.38, 5.3%), 3.61 (SD = 2.16, 7.1%) minutes on average for finding blocks in Squiral, Pong2, and GG2 respectively. We performed a Kruskal-Wallis test to determine if these differences in time spent searching blocks across semesters were significant. We did not find any significant difference between semesters on time spent searching blocks. Students in the experimental group did not spend more time searching for blocks than students in the control group. This indicates that searching for blocks during the lab
assignment did not help control group students more quickly find the blocks during the following homeworks. Therefore, block-based Parsons problems can save students time by eliminating the need for changing categories, looking through blocks in the category, and choosing the desired block.

**Edit Input.** We also investigated the time students spent on editing block inputs. Control group students need to type literals or characters into block inputs while solving a problem from scratch, but block-based Parsons problems either provide the value to a block, or only allow drag and drop from variables, so X19 students do not need to type anything to enter block inputs. To determine edit input time for the control group, we choose 9 seconds (the third quartile value of all edit time) as the maximum value for editing and capped any times that exceeds this limit. We calculated the time students spent on editing block inputs and we found that students in the control group spent 2.88 ($SD = 1.68, 10.8\%$ of the total problem solving time), 2.05 ($SD = 1.84, 6.2\%$), 2.53 ($SD = 1.19, 9.1\%$) minutes on average to edit block inputs in PolygonMaker, Pong1, and GG1 respectively. We found a statistically significant positive Spearman correlation between the proportion of time spent editing block inputs and solving PolygonMaker ($\rho = 0.55, p < 0.001$), Pong1 ($\rho = 0.88, p < 0.001$), and GG1 ($\rho = 0.53, p < 0.001$). For the homework assignments, we found students in the control group spent 2.70 ($SD = 3.50, 11.2\%$), 2.47 ($SD = 2.36, 5.2\%$), 3.37 ($SD = 2.35, 5.4\%$) minutes on average for editing block inputs in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.92 ($SD = 1.27, 10.1\%$), 2.58 ($SD = 1.72, 5.0\%$), 3.07 ($SD = 1.39, 6.0\%$) minutes on average for editing inputs in Squiral, Pong2, and GG2 respectively. The Kruskal-Wallis test did not show any significant difference between semesters on time spent editing block inputs. This indicates that block-based Parsons problems can save students time by providing the default values for blocks without increasing students’ time editing block input for the following homeworks.

**Test ‘irrelevant’ blocks.** We defined ‘irrelevant’ blocks as blocks that were used during problem solving but did not exist in a student’s final submission or a common solution. Students may spend time using a block but finally find that it is irrelevant to the solution and hence delete it. For example, a student may use ‘set x’ and ‘change x’ block, instead of ‘move’ block, to change the position of the sprite, but would eventually delete those blocks since they are irrelevant to the solution. To quantify this behavior, we investigated students’ programming traces and for each block interaction, we classified the block as irrelevant when it neither exists in the student’s submission nor exists in a correct solution. Then we accumulate the time each student spent interacting with irrelevant blocks in each assignment. Note that we excluded from this measure any time that students spent on editing the irrelevant block inputs or on creating a new block since these measures were already included in the previous two behaviors. We found that students in the control group spent 2.31 ($SD = 2.16, 8.7\%$ of the total problem solving time), 1.36 ($SD = 1.63, 4.1\%$), 1.46 ($SD = 1.58, 5.2\%$) minutes on average for testing irrelevant blocks in PolygonMaker, Pong1, and GG1 respectively. A Spearman’s rank-order correlation was run to assess the relationship between the time spent
on irrelevant blocks and solving the lab assignments. There was a statistically significant, very strong positive correlation between the proportion of time spent on irrelevant blocks and solving PolygonMaker ($\rho = 0.87, p < 0.001$), Pong1 ($\rho = 0.96, p < 0.001$), and GG1 ($\rho = 0.95, p < 0.001$). For the homework assignments, we found students in the control group spent 2.38 ($SD = 3.78, 9.1\%$), 2.06 ($SD = 2.22, 4.3\%$), 4.93 ($SD = 9.28, 7.9\%$) minutes on average with irrelevant blocks in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.33 ($SD = 2.09, 7.0\%$), 1.44 ($SD = 1.23, 2.8\%$), 2.20 ($SD = 2.01, 4.3\%$) minutes on average on irrelevant blocks in Squiral, Pong2, and GG2 respectively. The Kruskal-Wallis test revealed significant differences on time spent using irrelevant blocks between semesters in Squiral ($\chi^2(3) = 12.03, p = .007$). However, a post hoc Dunn's test show no higher usage of irrelevant blocks in the X19 semester compared to any other semesters. Additionally, there was no significant difference between semesters on time spent using irrelevant blocks in Pong2 and GG2. This indicates that students did not benefit from using the irrelevant blocks during the preceding lab. This is consistent with Harms et al’s study that irrelevant blocks (distractors) took students time without contributing to learning [Har16].

By adding the time spent on these three behaviors together, we found students spent 6.5 minutes on these unproductive behaviors on average in PolygonMaker (24.5% of the total problem solving time), 6.1 minutes (18.2%) on average in Pong1, and 6.7 minutes (23.8%) on average in GG1. Since there is no significant difference in the total time spent on homework assignments between the control and experimental groups, we found that students in the control group did not benefit from these behaviors. Furthermore, control group students did not spend less time than X19 students on these behaviors while solving homework assignments. This means that the time they spent on these behaviors during the lab did not save them time during the homework– they did not learn where the blocks were, get faster at editing inputs, or spend less time on irrelevant blocks. Hence we can argue that these behaviors are unproductive. The block-based Parsons problems save students time by preventing those unproductive behaviors without causing them to significantly increase later.

Our results in section 5.3.1 shows that Parsons problems save more time than we calculated here. We estimated the time spent on our three unproductive behaviors conservatively, but since they do not account for all the time differences between the experimental and control groups, control group students must have spent additional time exploring the problem solution space. For example, we observed traces where students repeatedly enter values in ‘move’ and ‘turn’ blocks and run the code to see how it draws, rather than trying to design a math- or pattern-based solution to draw a shape. To explore differences in how the groups explored the solution space, we designed a visualization to show how students’ programs changed over time via programming path traces. We hypothesize that the limited blocks in Parsons problems reduce the need for students to explore the problem solution space. If this is true, then a visualization of programming traces should show smaller differences between consecutive states for Parsons problems, and more similarity between Parsons problems traces. To investigate this, we sampled 15 snapshots, evenly distributed
throughout students’ programming traces, from each student’s PolygonMaker programming traces in F16, S17, F17, and X19 semesters. Figure 5.3 visualizes the 15 sequential snapshots of each student for PolygonMaker and Squiral. Each point on the graph represents a program snapshot, and the axes are a 2D projection that preserves tree edit distances between two snapshots\(^2\). Figure 5.3 (left) demonstrates that Parsons problems paths (yellow) have much less diversity than those from writing block-based code. Figure 5.3 (right) shows that this difference did not extend to the next homework assignment, Squiral. This shows that although X19 students did not highly explore the lab solution space, they still had similar programming paths to the control group on the subsequent homework.

![Figure 5.3](image)

**Figure 5.3** A 2D visualization of the similarity between students’ snapshots at different points in time, across semesters, for PolygonMaker and Squiral. Color indicates the snapshot’s semester, and black borders indicated a submitted solution.

Since Parsons problems provide only the necessary blocks for solving a problem, students will not spend time on irrelevant blocks, searching for blocks, or editing block inputs. The limited blocks in Parsons problems constrain students’ solution space exploration, which saves students time. However, reducing these behaviors does not seem to negatively impact homework performance, supporting our hypothesis that Parsons problems are more efficient than writing block-based code.

### 5.4 Discussion

Our results show that solving Parsons problems can lead to more efficient learning than writing block-based code. Our main hypotheses are supported by the results. We found students who do Parsons Problems instead of problem solving during programming labs: accomplish the lab assignment faster than previous semesters’ students (H1), perform no worse than previous semesters’ students on the following homework assignments, as measured by grades, time on task, and hint usage (H2) and the final programming project (H3).

\(^2\)We derived this layout using non-metric multidimensional scaling [CC00].
**Impact of Parsons Problems on Students’ Learning.** In this study, we found Parsons problems save students 43.6% to 65.2% of total problem solving time, without reducing performance on subsequent problems. This is consistent with previous studies that Parsons problems improve students learning efficiency [Eri17; Eri18; Har15]. Importantly, our results show that the benefits of Parsons problems can generalize to a 6-week classroom study and comparably larger problems (with up to 18 lines of code that took 33 minutes on average). Additionally, our results differ from prior work in that we compared Parsons problems directly with programming from scratch (without worked examples or tutorials), and we find just as strong results.

However, we did find some nuances to Parsons problems’ benefits. First, we saw X19 students had higher usage of hints, though not significantly higher, for Pong2 and Squirrel than students in other semesters. While X19 students performed just as well, we cannot determine whether this was in part due to a higher reliance on hints. However, due to the quasi-experimental nature of our study, we could not remove hints from the homework without creating an unfair comparison. Additionally, we saw some suggestive evidence that Parsons problems may actually improve performance on subsequent assignments (e.g. Pong2, GG2). This is agreed somewhat with results from Ericson et al. [Eri18], who found that adaptive Parsons problems led to higher learning than students who solve off-task Parsons problems on turtle graphics. Since our Parsons problems were not adaptive, this suggests that we may see measurably improved performance from Parsons problems if we employ Ericson et al.’s approach.

We did find a few significant differences between control semesters for lab assignments (e.g. time on task and class grades for PolygonMaker), and these differences may be explained by the variance of TAs and student across semesters. The teacher may also become better at teaching this course. However, the magnitude of the reduction in time on task we saw in the X19 semester is much larger than these fluctuations, and it unlikely that our results were due to chance.

**How do Parsons Problems Improve Learning Efficiency?** To investigate the efficiency of Parsons problems, we analyzed students’ programming traces and defined three unproductive behaviors that can be avoided by solving Parsons problems. A key contribution of our study was comparing Parsons problems to writing block-based code directly. Our results show that, despite the affordances of block-based programming that can improve students outcomes [PB15; WW17], such as drag-and-drop blocks and limited block sets, Parsons problems implementation in block-based environment still add some additional benefits. Our results suggest that Parsons problems may save time by preventing students from engaging in various unproductive activities such as searching blocks, editing block inputs, and testing irrelevant blocks. Again, while we cannot conclude that those activities do not contribute to learning, we saw no evidence that engaging in these activities in the lab assignments reduced their incidence in the subsequent homeworks. Interestingly, we found high (sometimes almost perfect) correlations between the proportion of time spent on these behaviors and total time spent on homeworks. However, we do not claim that Parsons problems reduce
these unproductive behaviors in later assignments, just that these behaviors are clear indicators of struggling students.

Even with the time saved for unproductive behaviors, we found that we could not explain all the time saved by Parsons problems. However, our results in Figure 5.3 suggest that one way Parsons problems save time is by reducing the space of possible partial programs that students must explore, constraining them along a more direct path towards a solution. Prior work suggests that novice programmers create a wide variety of solutions to programming problems [RK14], especially in block-based programming [Pri16]. By constraining the search space, we seem to save time. This also makes sense from the perspective of cognitive load and element interactivity [Swe10b]. Since students have fewer blocks to interact with, Parsons problems would impose less cognitive load to students. As Ericson hypothesized, Parsons problems may lead to lower cognitive load, although they did not find enough evidence to support this [Eri18].

5.5 Conclusion

Limitations. This is a quasi-experimental study with important limitations. First, our analysis involved comparisons of 3 variables (time, grades and hint usage) across 6 assignments, which involved repeated statistical tests. Because our analysis was targeted at investigating a priori hypotheses, we did not correct for these multiple tests, except to control the false discovery rate for post hoc tests. Second, we use six semesters data and the teacher may become more skilled in teaching this course and the variance of undergraduate TAs may have an impact on students performance as well. Third, even though the course is designed for non-majors and students are self-selected to register for this course, students' programming ability may vary from semesters. These are inherent limitations to any quasi-experiment across semesters; however, we mitigated this as much as possible by comparing across many semesters.

In conclusion, this study evaluates the effectiveness of Parsons problems for block-based programming and we found that Parsons problems save students nearly half of total problem solving time. This paper contributes to the design of block-based Parsons problems and provides evidence on the effectiveness of Parsons problems that mainly comes from solving the problem instead of from the benefits of the block-based programming language. Additionally, it provides insights on how Parsons problems impact students problem solving behavior. Our results encourage teachers to ask students to solve Parsons problems instead of writing code in novice programming environments. Furthermore, this study also increased our knowledge of how Parsons problems work.
CHAPTER

6

STUDY 4: TOWARD DATA-DRIVEN EXAMPLE-BASED FEEDBACK FOR NOVICE PROGRAMMING

This chapter was adapted from: Zhi, R. et al. (in press). “Toward Data-Driven Example Feedback for Novice Programming.” Proceedings of the 12th International Conference on Educational Data Mining. 2019. [Zhi19c]

This study introduces the development and evaluation of data-driven example-based feedback in Snap!. The high-level RQ3 is addressed in this Chapter.

Abstract

Viewing worked examples before problem solving has been shown to improve learning efficiency in novice programming. Example-based feedback seeks to present smaller, adaptive worked example steps during problem solving. We present a method for automatically generating and selecting adaptive, example-based programming feedback using historical student data. Our data-driven feature-based (DDF) example generation method automatically learns program features from data and selects example pairs based on when students complete each feature. We performed an experiment to compare three example generation methods: Student trace data, Data-Driven
Features (DDF), and Expert examples. Two experts rated the quality of feedback for each generator, and they rated both the Expert and DDF example feedback as significantly more relevant to students’ goals than the Student example feedback. However, there were no significant differences between the DDF and Expert examples. We compared these approaches to one that combined DDF with an Interactive Selection step (DDF-IS), where the user (in this case, an expert) selects their preferred data-driven feature before an example is selected. DDF-IS produced significantly more relevant examples than all other approaches, with significantly higher overall example quality than DDF. This suggests that our DDF approach allows more relevant examples to be selected than existing approaches, and that we may be able to leverage interactivity with the student to further improve example quality.

6.1 Introduction & Background

Prior studies show that worked examples are an effective instructional support to help novices learn complex tasks [Swe06; TR93; SC85; Ger04]. Sweller argues that “...for novices, learning via worked examples should be superior to learning via problem solving.” [Swe06]. In the domain of programming, researchers also suggest using worked examples to teach novices [CB07; Vih11]. Empirical studies have shown that interleaving worked examples with similar practice problems is more effective than solving only equivalent programming problems by writing code, as students spent less time on training tasks and performed better on a posttest [TR93]. However, worked examples are traditionally only offered to students in between problem solving attempts [TR93; Mor15; Eri17; Eri18], and they do little to assist students when they have difficulty during problem solving. Based on the idea of worked examples, researchers have explored example-based feedback [Gro14b; Coe17; Ich17], which shows a correct piece of code to help students learn during problem solving, as a form of adaptive, on-demand support [Gro14b]. Similar to a high-level on-demand hint, example-based feedback demonstrates one step in a correct solution to the problem the student is working on, selected adaptively to match the student’s code. Keuning et al. argue for the need for such feedback in their review on automated feedback generation for programming, saying “the very low percentage of tools that give code examples based on the student’s actions is unfortunate, because studying examples has proven to be an effective way of learning” [Keu18]. As Gross et al. argue, showing a relevant partial solution could impose less cognitive load and be easier to visually present to novices than showing a full solution [Gro14b]. Ichinco et al. found that novices have trouble transferring what they learned from similar problems to their own code [IK15]. Example-based feedback addresses this by providing example steps from the same problem the student is working on. Figure 6.1 presents a prototype of what example-based feedback might look like in a block-based novice programming environment. The example is presented with a “before” state, similar to the student’s current code, and an “after” state that completes a desired feature, helping
Existing example-based feedback systems may rely on a library of expert-authored examples [HB17; IK15; Coe17], which can be costly to maintain, as instructors are unlikely to create new examples [HB11]. Additionally, the example code may show an example solution to a related problem [HB17] that requires students to transfer knowledge to solve their current problem, which may be challenging for weaker students who need more help [Ich17]. To address these limitations, we present a data-driven method to create example-based feedback using historical student data. We focus on the domain of programming, where many problems have a vast space of possible solutions [RK17; Pri16], so a static set of examples is unlikely to be relevant to all students. To our knowledge, only Gross et al. have previously employed a data-driven method to derive examples adaptively to help students [Gro14b]. They found that examples derived from expert solutions are perceived by students as more helpful and help students make more solution improvements than those derived from student solutions. However, they simply presented the complete student solution which was most similar to the current student’s code (according to a distance metric), rather than trying to systematically identify and address students’ current programming goals. This suggests that existing evaluations of data-driven methods for example-based feedback have not explored the full potential of the approach. We hypothesize that with the innovations in this work, data-driven examples can be more adaptive than expert-authored ones, while still presenting correct and interpretable solution steps.

In this work, we present an approach to automatically detect students’ progress towards a solution, and suggest example-based feedback from historical student data. Our data-driven method first processes prior correct student solutions to discover meaningful features, labels the parts of code that contribute to each feature, and removes code that does not contribute to the solution, to
create simple data-driven code examples that contain only the code needed for each feature. We then automatically label the student’s current code with a “feature state” representing the presence or absence of each data-driven feature needed for a correct solution. We then adaptively select example feedback that contains the same features as the current student code, and adds a new feature that is relevant to solving the current problem. In contrast, other example feedback systems show code from a related similar problem, requiring students to study the example and transfer what they learn to the current context [HB17].

We evaluated two data-driven methods to generate and select example-based feedback for historical student hint requests: Data-Driven Features (DDF), and Data-Driven Features with Interactive Selection (DDF-IS). We compared these methods against two baselines: (1) Expert-authored examples (Expert) and (2) examples generated naively from correct student solution traces, showing the code added between consecutive test runs (Student). The data-driven features (DDF) algorithm cleans prior students solutions and generates example pairs where the “start code” has the same features as the student’s code, and the “end code” adds a new feature that is not yet present in the student’s code. The DDF with Interactive Selection (DDF-IS) approach explored the potential to improve algorithmic DDF example feedback selection by having a user interactively select the data-driven feature with which they want help, before the algorithm selects an example. To simulate this experience, we used an expert to select this feature, representing a best-case scenario for interactive selection.

We adapted a multidimensional data-driven hint evaluation rubric from previous work to evaluate the example-based feedback quality based on Relevance, Progress, Interpretability, and Similarity. Our findings showed both the Expert and DDF feedback were significantly more relevant to students’ goals than the Student feedback, but there were no significant differences between the DDF and Expert examples. This suggests that our DDF feedback can reasonably replace Expert-authored examples in situations where they are unavailable or difficult to scale. We also found that DDF-IS produced significantly more relevant examples than all other approaches, with the highest overall example quality, significantly higher than DDF. This suggests that in the best case, data-driven feedback may leverage interactivity with the student to further improve example quality.

The contributions of this paper are: 1) a data-driven algorithm capable of generating adaptive example feedback for students during programming, and 2) an initial evaluation showing that these adaptive examples can be more relevant than static, expert-authored examples.

6.2 Method

The goal of this work is to present and evaluate a data-driven feature-based (DDF) method for generating and selecting example-based feedback (explained in Section 6.2.2). To evaluate our DDF approach, we generated example-based feedback for historical student help requests, and asked
experts to evaluate the quality of each type of feedback, comparing against two baselines (explained in Section 6.2.3.1).

6.2.1 Dataset

Our dataset comes from iSnap [Pri17d], which extends the Snap! block-based programming environment with logging and on-demand, data-driven hint support. It logs student interactions with the system, including complete code snapshots after each edit. The data were collected during the Fall 2016 (F16), Spring 2017 (S17), and Fall 2017 (F17) semesters in an introductory computing course for non-majors, held at a research university\(^1\). In each semester, the students completed 3 in-lab assignments with access to help from teaching assistants and 3 homework assignments independently. In this paper, we selected one homework assignment Squiral for the example code generation and evaluation. In Squiral (shown in Figure 6.2), students program a “sprite” to draw a spiraling square-like shape using loops, variables, arithmetic operators and a custom block (function).

Common solutions for Squiral contain 7-10 lines of code. The original dataset contains 57 (F16), 43 (S17), and 47 (F17) Squiral assignment submissions. Since iSnap offers students on-demand, data-driven hints which may alter students' problem-solving patterns, we exclude students who requests hints in the dataset used for the example-based feedback generation. Our remaining data contained 38, 29, 39 code traces for the F16, S17, and F17 semesters, respectively. Each code trace contains the set of timestamped snapshots that comprise all of a student’s work on a problem.

6.2.2 Data-driven Example-based Feedback Generation

We propose an algorithm to automatically generate data-driven, example-based feedback with the following high-level steps: 1) Extract a set of data-driven features from prior student code traces, which each describe a property of a correct solution. 2) Use the features to clean correct student

---

\(^1\)All datasets are available at https://pslcdatashop.web.cmu.edu/Project?id=321
code traces by removing extraneous code that does not contribute to a correct solution. 3) Generate pairs of example code snapshots that demonstrate how to complete a single feature. 4) Choose an appropriate, personalized example pair upon student request.

Our goal in this work is to perform a preliminary evaluation of our algorithm before implementing a user interface and evaluating its impact in practice. However, to contextualize our work, Figure 6.1 presents one way that such feedback could be presented to a student in iSnap when they request help. Our DDF example pairs consist of “start code,” similar to the student’s, and “end code” that demonstrates how to complete a single feature. Our design draws inspiration from a variety of theoretical and empirical sources, including cognitive load theory [Swe98], Vygotsky’s Zone of Proximal Development [Vyg78], worked examples [Swe06], learning from subgoals [Mor15], and compare/contrast tasks [Pat13].

Research on worked examples suggests that seeing examples which break a problem down into sequential steps (e.g. features) can be a more efficient way of learning than problem solving [Swe06; TR93; SC85]. According to cognitive load theory, worked examples are effective because they lower the extraneous cognitive load (mental effort) imposed by the instructional materials. We designed our examples to present steps as a pair of “start” and “end” code, as previous studies have shown that comparing and contrasting examples is an effective learning activity in many domains [Pat13; Sch11]. The example is adaptively selected to keep students in the Zone of Proximal Development [Vyg78] by starting with code similar to the student’s, which they can already understand, and scaffolding the completion of a new feature, which they cannot yet accomplish on their own. The work of Morrison et al. [Mor15] suggests that programming examples that are broken into subgoals can help improve learning for novices. When examples are isomorphic to the problem solving task, as in our case, they found that it is most effective for students to label subgoals themselves, a feature we could easily incorporate into our example-based feedback.

6.2.2.1 Step 1: Data-driven Feature Generation

The goal of an example pair is to present how a meaningful self-contained portion of solution code, or feature, can be completed. An assignment may have students program multiple features, and the final correct solution should have all the correct features present. For example, in Squirrel (shown in Figure 6.2), a feature could be to move the sprite in a square shape, draw some figure on the screen, or repeat the spiral the correct number of times. In our previous work [Zhi18b], we manually defined expert-authored features (shown in Table 6.1) in a systematic way, and we also implemented a data-driven algorithm to automatically identify code features from student solutions. Our results showed that many of the data-driven features were easily interpretable and closely matched the expert-authored features. The two methods also had moderate agreement on whether a given student was in the same state or different states.
Table 6.1 Expert Features and Corresponding Data-Driven Features, as derived in [Zhi18b].

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Brief Description</th>
<th>Data-driven Analogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1. Procedure</td>
<td>Primary code inside of a procedure.</td>
<td>D1: Create a procedure OR a variable.</td>
</tr>
<tr>
<td>E2. Draw Anything</td>
<td>Able to draw anything on screen.</td>
<td>D8: Use a 'repeat' AND create a variable OR a parameter.</td>
</tr>
<tr>
<td>E3. Move ‘Square-like’</td>
<td>Able to move sprite in a square-like fashion.</td>
<td>D11: Have a 'move' AND a 'turn' in a 'repeat' AND have a 'pen down.'</td>
</tr>
<tr>
<td>E5. Repeat Correct # of Times</td>
<td>Repeats square-like movement correct number of times.</td>
<td>D5: Have a ‘multiply’ block with a variable OR two nested ‘repeats’.</td>
</tr>
<tr>
<td>E6. Move ‘Variably’</td>
<td>Movement is based on a variable not literal amount.</td>
<td>D10: Have a 'move' with a variable argument inside of a 'repeat'.</td>
</tr>
<tr>
<td>E7. Move ‘Squirally’</td>
<td>Increase length to move for each side.</td>
<td>D7: Change a variable inside a ‘repeat’.</td>
</tr>
</tbody>
</table>

The full procedure for data-driven feature extraction is given in [Zhi18b], but we outline its high-level steps as follows:

1) **Preprocess student solutions:** Some student solutions may contain extraneous code or procedures that were used for testing or resetting the environment, which we attempt to remove before extracting features. Specifically, we used the SourceCheck algorithm [Pri17a] to identify and remove whole scripts and procedures that do not match any element of an expert-authored solution in the correct student solutions.

2) **Generate code shapes:** To identify common code patterns in correct student solutions, we extract a set of code shapes, or syntactic structures, from the solution code by converting students’ solution code into abstract syntax trees (ASTs) and then identify all pq-Gram subtrees [Aug10] in each AST, to form our initial set of “code shapes.” This includes all code shapes from all correct solutions.

3) **Remove duplicates:** The initial set of code shapes may include very similar shapes, including some AST patterns that are subsets of others. Therefore, we remove these duplicates by measuring the co-occurrence of code shapes in all student code traces and keeping only the more specific code shape and discard the other duplicate if two code shapes almost always appear together in the same code.

4) **Identify decision shapes:** Due to varied problem solving strategies, some code shapes may not appear in every correct solution. For example, a Squiral solution can either use nested repeat or a single repeat block to rotate the correct number of times (as shown in Figure 6.2), but not both. Therefore, we define a decision shape as a disjunction of code shapes, where almost all solutions...
contain exactly one of the component code shapes. A decision shape is present in a solution if any one of its code shapes exist in its code.

5) **Filter out uncommon code and decision shapes:** Since we are interested in using code and decision shapes to represent features of a correct solution, we keep only those shapes which appear in the vast majority of correct solutions, and filter out the rest.

6) **Form features:** Our goal is to define a small set of features that collectively represent a complete solution. However, the previous steps will generate tens to hundreds of code and decision shapes for a relatively simple problem like Squirrel. We therefore combine these smaller shapes into larger features using a form of hierarchical clustering. We iteratively combine any two features that most frequently co-occur across student data until the size of the observed state-space defined by the features starts to decrease rapidly.

7) **Represent student code by feature vectors:** Once the features are formed, we can represent a student’s current code as a vector indicating the presence or absence of each feature. A student starts with a feature state of all 0s, and a correct solution should have all features present, resulting in a vector of all 1s.

### 6.2.2.2 Step 2: Cleaning Student Code

To extract good example pairs from prior student traces, we need to first remove excess blocks which do not contribute to a correct solution, as these may distract students and make examples harder to interpret. Identifying the excess blocks can be difficult, especially for intermediate partial solutions, since students can construct solutions in a large variety of ways [Pri17a; RK17]. We address this by leveraging the features we defined in step 1 to exclude *irrelevant code*, which does not belong to any feature. Specifically, our cleaning procedure removes one node from the abstract syntax tree at a time (including all its children), then checks whether removing this node causes a currently completed feature to become incomplete. If removing a node “breaks” a feature, we assume that it is necessary and add it back; otherwise, we remove it. We also check to make sure removing the node does not break any code dependencies, such as deleting a variable declaration when the variable is used elsewhere. We iterate over every node in a recursive, breadth-first manner, starting from the root node. Once this iteration stops, it produces a cleaned partial solution, where all irrelevant code has been removed. We apply this cleaning procedure to all snapshots in correct solution traces. With well-defined features, this process can effectively clean a large variety of both partial and complete solutions, ensuring that all remaining code is useful. We use this process both for cleaning code and for extracting example pairs, as described in the next step.
6.2.2.3 Step 3: Extract Example Code

Our goal is to create a database of correct, meaningful, and self-contained example pairs to offer as feedback to students. Naively, we could extract a single example pair for each feature from each correct code trace, since each student completed each feature at least once. However, we want to generate as many example pairs as possible, so that the algorithm can adaptively select one that is similar to another student’s code. We therefore developed a method to generate many “synthetic” example pairs, each consisting of a pair of code states \((c_0, c_1)_i\), from any cleaned student code trace. Recall that each pair should cleanly demonstrate the completion of exactly one feature by contrasting a “start code” state \((c_0)\) and an “end code” state \((c_1)\). The algorithm first extracts one example pair each time a student completes a feature \(f_i\), with \(c_1\) defined as the snapshot right after \(f_i\) was completed, and \(c_0\) as the snapshot right after the prior feature \(f_{i-1}\) was completed. We generate additional example pairs from each cleaned snapshot in a student solution trace with the following procedure. For each snapshot, the algorithm labels it as an end state, \(c_1\). It then removes exactly one data-driven feature from \(c_1\) to create a \(c_0\), and together these form the example pair. This feature removal is accomplished using the code cleaning procedure described above; however, instead of removing irrelevant nodes, we use it to remove whole features. The algorithm first tries to remove one leaf node, \(l_i\), at a time. Since the snapshot has already been cleaned, removing this node will either create an invalid code state, or cause a feature to become incomplete. In the later case, the cleaning procedure was run to remove all other code associated with the removed feature. The resulting cleaned code becomes the \(c_0\) for the example pair, and our cleaning procedure guarantees that \(c_0\) will have exactly one less feature than \(c_1\). The \((c_0, c_1)_i\) pair is added to a list which stores our example pairs. The algorithm then repeats this process recursively on \(c_0\), which becomes the \(c_1\) for new example pairs, until no new pairs can be generated. In this case, we generate many example pairs per snapshot in a solution trace. While some are redundant, many are unique.

6.2.2.4 Step 4: Select an Example Code Pair

When a student requests help, we aim to provide them with the most appropriate example pair in our database as feedback. We define two ways that we can identify this example pair: 1) a Data-Driven Features (DDF) approach, using an algorithm to select the best example pair, or 2) an Data-Driven Features with Interactive Selection (DDF-IS) approach, giving the student the information needed to select an example pair. We first consider the DDF approach, in which we attempt to select the example pair which is most similar to the current student’s code. Based on this selection criteria, the selected example code pair should be very similar to the student’s code, with the goal of minimizing the effort needed to process the “start code” and allowing the student to focus on the feature demonstrated by the example. In this study, for the DDF feedback, we selected an appropriate example code pair as follows:
Generate proper example pair candidates: To filter out inappropriate pair candidates based on the student's completed features, we select only those example pairs whose “start code” has the same features as the current student code, from the example pair lists (generated in Section 6.2.2.3). If we cannot find any, we sort the example pairs by the Hamming distance between the feature states of “start code” and student's current code. Then we select the example pairs with the closest start-state to student’s current code. If there are multiple example pairs available (they all have the same start state), we use the SourceCheck algorithm to sort the example pairs based on the similarity between the “start code” and student's current code.

Select one example pair from candidates: We iterate over the example pair candidates and select exactly one pair, which accomplishes a feature that the student has not finished yet, preferring features that the majority of students in a state similar to the current student will take next.

To generate interactive example-based feedback, we need a way to communicate to the student which features have an available example that they can request. By default, our features are unlabeled (having been generated automatically from data), but with a small amount of instructor effort (about 3 minutes), they can be labeled. By viewing the code shapes required by each feature, an instructor familiar with the problem can generate a short, human-readable description, such as “move the sprite using a variable”, or “make the sprite move further each time”. These options can then be shown to a student. Once the student has selected a feature for an example, we select an appropriate example pair that accomplishes that feature with start code matching the student’s current feature state. If there are multiple options, we use the same criteria as in DDF: select the example pair with the most similar start state to the student's code, which contains a proper subset of the student's features. Note that unlike in DDF, a student could request an example for a feature that they have already completed. Students might decide to do this when they are unsure if they have completed a feature correctly and want to see an example for confirmation.

6.2.3 Expert Evaluation

To test the feasibility of our method before building the whole system and conducting a user study, we did a preliminary expert evaluation of our algorithm. We generated example pairs to support student code snapshots from our historical dataset and evaluated their quality. To simulate real student help requests where an example might be needed, we selected snapshots that corresponded to times when historical students requested hints from iSnap. As in prior work on evaluating feedback [Pri17c], we sampled up to two hint requests (and their corresponding student code snapshots) from each student. We sampled 50 hint requests in total including 20 in F16, 20 in S17, and 10 in F17. For each hint request we generated four example code pairs using different techniques: DDF, DDF-IS, and our two baselines, Student and Expert (explained in the next section). For the examples derived from student data (DDF, DDF-IS, Student), we generated examples using semester-
based 3-fold cross-validation. For each semester, we used the other two semesters’ data as training data to generate the examples. We derived 9, 9, and 10 data-driven features for F16, S17, and F17, respectively.

Two co-authors, who neither authored examples nor worked on the algorithm itself, served as experts to evaluate the generated example pairs. Both experts have extensive experience in Snap! and the Squirrel assignment. We built an interface in iSnap to present each expert with student original code and the example pair. Then we asked the experts to assess the example pair based on a detailed example code rating rubric\(^2\), adapted from [Pri17c]. Our rubric has 4 attributes, each rated 1, 2 or 3, with higher scores being better. These 4 attributes measured: 1) **Relevance**: how relevant the suggested example code pair is to the student’s current goals, 2) **Progress**: how well the example code pair help students make progress towards the final correct solution, 3) ** Appropriateness & Interpretability**: how likely a tutor will be to suggest this example pair to a student and how easily a novice could understand the intention of the suggested example pair and 4) **Similarity**, how similar is the “start code” to the student’s code. The first 3 attributes are means to assess the quality of the example. The 4th is meant to help us understand the relationship between example similarity and quality, since all examples were selected based on their similarity to student code. During evaluation, experts had access to students’ code history, and based their ratings on the student’s individual context.

To ensure that the two experts had a similar understanding of the rubric, they rated 10 examples together, which were not used in this study. They then rated the 200 examples pairs used in this study in 2 rounds of 100 each\(^3\). In Round 1, they independently rated 40 example pairs and then discussed their ratings to resolve any conflicts and reach consensus. Their inter-agreement reliability across the 40 example pairs achieved squared-weighted Cohen’s kappas of 0.94, 0.91, 0.82, 0.90 for Relevance, Progress, Appropriateness & Interpretability, and Similarity, respectively, indicating very strong agreement. They then split the remaining 60 pairs and rated them individually\(^4\). In Round 2, one expert rated the remaining 100 pairs individually, and the other expert rated 75 of these, which were discussed until consensus was reached. Across the 115 example pairs that were rated by both experts, the total squared-weighted Cohen’s kappa was 0.77.

### 6.2.3.1 Baselines

We compare both the interactive and non-interactive data-driven example-based feedback against two baselines: Expert-authored examples and examples generated naively from Student data. For data-driven example-based feedback, we use the two strategies (DDF and DDF-IS) described in

---

\(^2\)Available at: http://go.ncsu.edu/edm2019-rubric. The rubric can be also found in Appendix B.

\(^3\)Due to the order in which examples were generated, Round 1 included DDF and Expert examples, and Round 2 contained Student and DDF-IS examples. However, raters were blind to the condition of the example.

\(^4\)The rated examples were in DDF and Expert conditions.
section 6.2.2.4. Our goal for the **Expert baseline** was to reflect a straightforward way of generating example-based feedback using a small number of expert-authored example pairs, which an instructor might reasonably create. This corresponds to the Next Step of the Nearest Sample Solution (NSNNS) strategy introduced by Gross et al. [Gro14b], which they found to be optimal. This strategy selects the next step of the nearest expert solution. To generate expert example pairs, two experts (specifically, two co-authors who did not rate the example pairs during evaluation) manually authored example pairs for *all* steps in the most common solution paths, which at least 10% of students took to solve the problem. Each of the two co-authors created the example code separately, based on the understanding that the example code should be useful to students and no extra explanations will be used to help students understand the suggested example code. During this process, the experts could review as many students’ code (including all the history) as they needed. Afterwards, they met and discussed all the example pairs that they authored and came to consensus on the example pairs. In the Squirrel assignment, the common solution graph has 14 nodes and 13 edges, of which 7 were on a primary solution path and 6 were on two alternative paths. To select an Expert example-pair for a student, we first used the SoureCheck algorithm to sort the Expert example pairs based on the size of the “start code” and the similarity between their “start code” and the student’s current code. We select the example pair whose “start code” accomplishes fewer features than the student’s code and is very similar to the student’s code.

It may seem unfair to compare the Interactive Data-driven examples to (non-interactive) Expert examples. However, we note that it would not be reasonable to create an *Interactive*-Expert baseline. Since the Expert examples consist of a small number of hand-authored examples, which completed features in a specific order (e.g. Feature 1 is *always* completed before Feature 2), it is not reasonable to give a student the option of selecting a desired example. If the student has only completed 2 features, they could simply select the example for the final feature and see a full solution. Since the goal of example-based feedback is to show only a single, incomplete feature, we always selected the closest Expert example-pair to the student’s current code. By contrast, our data-driven approach generates enough example-pairs that we are able to provide many choices of features to complete, without revealing any other features in the process.

Our goal for the **Student baseline** was to reflect a naive approach to extracting examples from student code, without the feature-based cleaning and selection of our own algorithm. Gross et al. [Gro14b] defined their baseline of student-derived examples to show only students’ submitted solutions, essentially giving away the whole answer. Instead, our baseline extracts multiple examples from students, based on when they ran their code. We hypothesized that students often run their code when they have completed a meaningful feature, making this a meaningful way to demarcate examples. We extracted examples from all correct student solution traces that did not request help. We selected student code snapshots based on when they ran their code. We treated consecutive run events within 15 seconds of one another as a single run event, and we took the last of these
events as the boundary between example pairs. For each consecutive pair of run events (more than 15 seconds apart), we extract an example pair consisting of the two corresponding snapshots. The code snapshot that happened earlier serves as the “start code” of the example pair, and the one happened later serves as the “end code”. When selecting an example pair to show as feedback for a given student, we used the SourceCheck algorithm to find the nearest “start code” and present that example pair as feedback.

6.3 Results

We structured our analysis around the following research questions: **RQ1**: Can we create useful example-based feedback naively from student data, without a data-driven algorithm? **RQ2**: How does the quality of data-driven example-based feedback compare with that of expert-authored example-based feedback? **RQ3**: Can the quality of data-driven example-based feedback be improved if an example is selected interactively, rather than automatically?

We address each RQ by comparing the quality of example-based feedback generated by four feedback approaches explained above: 1) Expert-authored (Expert), 2) Naive Student Data (Student), 3) Data-Driven Features (DDF), and 4) Data-Driven Features with Interactive Selection (DDF-IS). We evaluated quality in terms of Relevance, Progress, and Interpretability, as explained above. For RQ1, we hypothesized that our results would be consistent with Gross et al. [Gro14b] that naively extracted student examples would not lead to high-quality feedback. For RQ2, we hypothesized that data-driven example-based feedback would be more relevant to the student’s code than expert feedback and just as useful otherwise. For RQ3, we hypothesized that interactive selection would improve the quality of data-driven example-based feedback.

6.3.1 Feature Coverage

From 106 student solution traces, the DDF algorithm was able to generate 13,927 unique data-driven examples, as shown in Table 6.2. Of these, only 728 (5%) were derived directly from a student’s trace, and the rest were generated with the recursive algorithm explained in Section 6.2.2.3. It took around 10 minutes (645 seconds) to generate all the data-driven example pairs. Table 6.2 shows the number of examples generated by each algorithm. It also gives the “snapshot coverage” and “hint request coverage” of each algorithm. The former refers to the percent of all observed snapshots which had an available example in the same feature-state (meaning the example started with the same set of features completed). For this calculation, we used the expert-authored features defined in [Zhi18b]. Hint request coverage considers only the 50 hint request snapshots we evaluated. These numbers are averaged across the 3 semesters.
Table 6.2 Total number of generated example pairs, corresponding average snapshot coverage and hint coverage in the expert-defined feature space.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DDF &amp; DDF-IS</th>
<th>Student</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td># of examples generated</td>
<td>13,927</td>
<td>242</td>
<td>13</td>
</tr>
<tr>
<td>Snapshot coverage</td>
<td>0.843</td>
<td>0.670</td>
<td>0.709</td>
</tr>
<tr>
<td>Hint request coverage</td>
<td>0.821</td>
<td>0.709</td>
<td>0.687</td>
</tr>
</tbody>
</table>

6.3.2 Expert Ratings

Our dataset consists of 50 hint requests and 200 example pairs (one example pair per feedback approach for each hint request). All the example pairs were rated on four attributes on a scale of 1-3: Relevance, Progress, Appropriateness & Interpretability, and Similarity, as described in Section 6.2.3. We found the first three attribute ratings showed significant positive pairwise Spearman correlations ranging from 0.51 to 0.81 (all $p < 0.001$). Similarity also had a lower, positive correlation with the other attributes, ranging from 0.24 to 0.34 (all $p < 0.001$). Due to the high positive correlation of the Relevance, Progress, and Appropriateness & Interpretability attributes, we also compute a Quality attribute, which sums all three attributes, with scores ranging from 3 to 9. Because all 4 example-based feedback approaches selected examples using SourceCheck’s code similarity function, we also investigated the relationship between SourceCheck’s calculated similarity and the expert-rated Similarity. We found a significant, positive correlation ($\rho = 0.29; p < 0.001$), suggesting that the similarity function is reasonable but could be improved. Table 6.3 reports mean values of each attribute for each example-based feedback approach.$^5$

Table 6.3 Mean attribute ratings (with standard deviation) for example pairs in from each approach.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DDF-IS</td>
<td>2.62 (0.73)</td>
<td>2.46 (0.76)</td>
<td>2.22 (0.82)</td>
<td>7.30 (2.05)</td>
<td>2.22 (0.86)</td>
</tr>
<tr>
<td>Expert</td>
<td>2.24 (0.89)</td>
<td>2.36 (0.80)</td>
<td>2.14 (0.90)</td>
<td>6.74 (2.24)</td>
<td>2.18 (0.80)</td>
</tr>
<tr>
<td>DDF</td>
<td>2.12 (0.92)</td>
<td>2.06 (0.89)</td>
<td>1.84 (0.82)</td>
<td>6.02 (2.38)</td>
<td>2.28 (0.88)</td>
</tr>
<tr>
<td>Student</td>
<td>1.72 (0.90)</td>
<td>2.12 (0.90)</td>
<td>1.80 (0.86)</td>
<td>5.64 (2.20)</td>
<td>2.24 (0.87)</td>
</tr>
</tbody>
</table>

$^5$Although attribute ratings are ordinal, we report the mean and SD, since the median values for each attribute are generally the same (2 or 3)
To address our research questions, for each attribute we used Kruskal-Wallis test to determine if there was a significant difference in ratings across feedback generation approaches. For the overall Quality attribute, we found a significant difference among conditions ($\chi^2(3) = 16.06, p < 0.001$). We performed a post hoc Dunn’s test with Benjamini-Hochberg correction for multiple comparisons\(^6\) [BH95; Dun64] to identify pairwise significant differences between approaches. This showed a significant difference between Expert and Student examples ($z = 2.48, p = 0.026, r = 0.25$\(^7\)), DDF and DDF-IS ($z = 2.76, p = 0.017, r = 0.28$), DDF-IS and Student ($z = 3.69, p = 0.0013, r = 0.37$), suggesting that for overall quality, Student < DDF < DDF-IS, and Student < Expert.

We then inspected the difference for each individual attribute. A Kruskal-Wallis test showed a significant difference among approaches for the Relevance ($\chi^2(3) = 24.45, p < 0.001$). A post-hoc test using Dunn’s test with Benjamini-Hochberg correction showed a significant difference between DDF and Student ($z = 2.14, p = 0.049, r = 0.21$), Expert and Student ($z = 2.79, p = 0.016, r = 0.28$), DDF-IS and DDF ($z = 2.76, p = 0.011, r = 0.28$), DDF-IS and Student ($z = 4.90, p < 0.001, r = 0.48$), DDF-IS and Expert ($z = 2.11, p = 0.04, r = 0.21$), but we did not find a significant difference between DDF and Expert ($z = 0.65, p = 0.515, r = 0.07$). This suggests that for Relevance, Student < DDF = Expert < DDF-IS.

We also found a significant difference among conditions for the Appropriateness & Interpretability ($\chi^2(3) = 8.98, p = 0.030$). However, a post-hoc test using Dunn's test with Benjamini-Hochberg correction did not find any pairwise significant differences between conditions. For the other two attributes, similarly, we did not find any significant difference between conditions: Progress ($\chi^2(3) = 7.14, p = 0.068$); Similarity ($\chi^2(3) = 0.62, p = 0.89$). The results suggest that DDF-IS example pairs are more relevant to student code and are equally helpful and interpretable compared with expert-authored examples.

### 6.3.3 Inspection of Examples

To better understand our quantitative results, we manually inspected examples generated by each approach to understand how they differed and how those differences impacted the expert ratings. We investigated the following questions and present situations that highlight possible answers:

**Why does the naive Student baseline have low quality?** Our quantitative results show that our naive Student baseline does not produce useful example pairs. We manually investigated some Student example pairs to better understand why. Recall that the Student baseline extracts examples showing the changes between consecutive run of student code. We derived 242 of these example pairs across 3 semesters from 39 correct student submissions that did not request hints. Upon inspection, it is clear that this segmentation approach did not always produce meaningful example

---

\(^6\)We report p-values corrected with the Benjamini-Hochberg procedure to control the false discovery rate at 0.05, keeping the \(\alpha\) significance threshold at 0.05.

\(^7\)The effect size \(r\) is calculated as described in [RK16]
pairs that showed a meaningful solution step. For example, a Student example pair might simply rearrange the order of some code blocks. This is a common debugging behavior, but it does not usually yield a useful example. We also found that even when Student example pairs demonstrated a meaningful step, they were often selected for students who had already completed that feature. This is not surprising, since the Student baseline selected the example with the most similar start code, but this does not guarantee the student will not have additional features completed. Lastly, many Student examples contained extraneous code that made them harder to interpret. These three problems – lack of meaningful steps, repeating completed steps, and extraneous code – are all addressed in our DDF-based example selection. With good features, we can identify meaningful example steps, ensure that all examples complete new features, and remove extraneous code.

When were data-driven examples more adaptive than Expert examples? Our quantitative results suggest that the DDF-IS approach was able to generate examples that were significantly more Relevant than those of the Expert approach. We hypothesized that this was enabled by the large number of unique example pairs generated by DDF (13,927). By contrast the Expert approach used only 13 examples pairs, which covered the common solution paths that at least 10% of students took to solve the problem. While some of DDF-IS’s improvement may have come from its Interactive Selection step (explored below), our results in Section 6.3.1 also show DDF examples have larger coverage than the expert-authored examples for both the hint requests and student snapshots. Our manual inspection uncovered a number of situations when a relevant expert Example was not available, but a DDF example was. For example, a strong minority of students solved the Squirrel problem by using a second input (parameter) to store the side length of shape. Figure 6.3 shows that for the same hint request, the DDF-IS can suggest examples not only similar to the student code, but also relevant to what the student is working on. However, the expert-authored example only has one parameter in the custom block and suggests something that the student has already done. The Expert example becomes less helpful and irrelevant when a student has solution that deviates from the Expert examples. However, our data-driven examples are more adaptive in this scenario and can provide examples both similar and relevant to the student. We note that this adaptivity was enabled in large part by the recursive example generation algorithm outlined in Section 6.2.2.3. Only 2 of the 50 examples selected by DDF-IS were extracted directly from student code traces (the “naive approach”); the other 48 were generated synthetically.

When did DDF-IS select better examples than DDF? Since DDF and DDF-IS were selecting examples from the same pool of examples, but DDF-IS has significantly higher overall quality, we wanted to understand where the selection algorithms differed. We investigated some pairs and focused on when our algorithm failed to select relevant examples. In those scenarios, we found that the DDF would suggest the student to add a ‘pen down’, or add a custom block in the main script area. Those features are necessary for a correct solution, but they may have lower priority compared with other features such as “control the sprite to draw a square” or “repeat the spiral the correct
“number of times.” Additionally, we also found that the data-driven feature detector sometimes counted a feature as complete when it actually still had a bug or a missing block. Thus, it would suggest example pairs that accomplish new features without fixing the broken one. In the opposite case, the student may have already finished a feature, but the feature detector sometimes failed to detect it, showing a redundant example. This occurred frequently when students accomplished a feature in a unique way, not captured by our data-driven feature definition. For example, a few students attempted to use a recursive approach to solve Squiral, but since this was quite uncommon, our data-driven features failed to generate meaningful example pairs for this solution. The DDF-IS can help resolve the first two cases, since students can choose whether they need help on a feature that the DDF failed to detect.

6.4 Discussion

RQ1: Can we create useful example-based feedback naively from student data, without a data-driven algorithm? Our results suggest that naively extracting examples from student code based on when students ran their code does not yield high-quality example-pairs. These naive Student examples were lowest or second-lowest rated on every dimension, with significantly lower Relevance than all other approaches and significantly lower overall Quality than the DDF-IS approach and the Expert-authored examples. This is consistent with our hypothesis for RQ1. It is also in agreement with the work of Gross et al. [Gro14b]., who found that using student solutions as example-based feedback were rated lower by experts and led to less improvement in students than other, expert-authored examples. However, rather than using a student’s submitted code as an example as Gross et al. did, we used the changes that students made between running their code. We believe that this represents a more reasonable student baseline, since it only shows a part of the answer like the other feedback approaches. It should also theoretically limit the cognitive load needed for students to learn from the examples. Our results show that even so, a naive approach does not create high-
quality student examples, compared to other approaches. Our data-driven approach attempts to address this by creating examples based on when students completed features.

One of the goals of using student data to generate examples is that they can more closely match the student's current code. In fact, we were able to extract 242 unique Student examples, compared with 13 for the Experts. This did enable our system to identify Student examples that were more Similar to the help-requesting student's code than Expert examples (though not significantly more). However, it is also clear that Similarity alone did not translate into Relevance, since the Student examples had the lowest Relevance scores. We cannot guarantee that students make meaningful changes between two consecutive snapshots, and thus the generated example pairs may not accomplish a meaningful chunk of code. For example, the Student example pair may suggest something that a student has already done. In this case, we could use the data-driven features to identify a later snapshot with more relevant changes as the “end code.” This suggests the need for a more deliberate, data-driven process that can still create a large number of examples to select from without sacrificing quality.

**RQ2: How does the quality of data-driven example-based feedback compare with that of expert-authored example-based feedback?** We hypothesized that our Data-Driven Features (DDF) examples would be more relevant to help-requesting student's code than the expert baseline, with similar levels of Progress and Interpretability. However, our results did not support this hypothesis. There were no significant differences between the two approaches, with the DDF algorithm performing similarly on Relevance and slightly worse on Progress and Interpretability. Our original hypothesis was based on the premise that data-driven examples would be more Relevant, since they are selected from a much larger database of examples (4,000 to 5,000) that represents a large variety of student code. While this was not confirmed, our results do suggest that our current, non-interactive data-driven algorithm performs significantly better than naive Student examples and only marginally worse than Expert examples. Since it is often difficult to get instructors to author examples [HB11], this suggests our DDF examples should be a reasonable substitution.

There are a few possible explanations for the poorer performance of DDF. First, it is possible that Squirrel problem we analyzed may not have been complex enough to necessitate generating a large variety of examples, which is one of the key proposed advantages of our algorithm. If most hint-requesting students performed the steps of the problem in the same order as the Expert examples, these Expert examples would generally be Relevant, leaving less room for the DDF examples to improve upon them. We have some support for this explanation, since across semesters 53.8% to 85.7% of the hint requests we analyzed had an expert feature state that matched one of our Expert examples exactly. However, we also note that the expert baseline was rated less than 3 for Relevance 46% of the time, so there was clearly room for improvement, as we discuss later.

It is also clear that the DDF approach did not perform as well as expected, scoring between the Expert and Student baselines for Progress and Interpretability. There are two possible explanations...
for this: either the algorithm is failing to generate good example pairs, or it is generating useful example pairs, but failing to select the best pair to show a given student. The higher scores of the DDF-IS algorithm, which generated the same example pairs as DDF but allowed a human to select the best one, suggest that the problem lies in the DDF algorithm's selection of example pairs. This selection process (explained in Section 6.2.2.4) primarily uses the SourceCheck algorithm to identify the example with the most similar start state to the student's code. While this approach did lead to DDF having the highest Similarity ratings, there was only a $\rho = 0.238$ correlation between Relevance and Similarity. This suggests that Similarity alone is not a good proxy for example Relevance or overall quality. The challenge of automatically selecting an appropriate example also reflects prior work on data-driven feedback [Pri18], where low-quality hints arose because the hint generation algorithm was unable to identify the most useful hints and filter out less useful hints.

**RQ3: Can the quality of data-driven example-based feedback be improved if an example is selected interactively, rather than automatically?** Our results show that the quality of data-driven example-based feedback can be improved if the examples are selected interactively. The DDF-IS algorithm scored significantly higher than the DDF algorithm on Quality and Relevance, even significantly outperforming the expert baseline on Relevance, with the best overall scores. Recall that for the DDF-IS examples, we manually tagged the data-driven features with natural language labels, which a student could reasonably select from when making an example request. In this preliminary evaluation, we simulated this selection process by manually selecting the best data-driven feature to show to each student. Importantly, we only selected the feature (e.g. “move the sprite using a variable”), not the example, and the algorithm still selected the best possible example for a given feature (from hundreds of possible choices). Still, it is likely that an expert familiar with the system is more likely to choose an appropriate feature than a student, so this represents an upper bound for how well an interactive, data-driven example selection algorithm could reasonably perform. These results show that, when this proper selection is applied, the larger database of example pairs generated by our algorithm can lead to more Relevant examples than a static set of expert-authored examples. This represents a growing trend among data-driven systems that support programming to leverage human (i.e. student and instructor) knowledge in conjunction with data-driven algorithms [Hea17; Pie15; Wu18]. However, from our preliminary expert evaluation, we can not make claims about how students would actually react to such a choice. Our results also suggest ways that we can improve the automated example selection procedure. The majority of low-quality DDF example pairs would select a feature that the student needs but could be less relevant than other features. For example, one example pair suggests adding the student’s custom block into the main script so they can test it. However, in Snap!, students can click on the custom block to test it directly. This feature should therefore have lower priority than others. For the example shown in the Results section, the data-driven feature detector failed to detect correct variations in student code, and suggested a feature that the student had already completed. These examples suggest that it may be more
important to select the most important feature to demonstrate, before considering the similarity of the example to the student’s current code.

6.5 Conclusion

This study presents a method to generate data-driven examples automatically from historical student data. We evaluated two versions of data-driven generated examples using the student code when they need help, compared with two baselines: examples authored by experts and examples derived naively from student solution traces. Experts evaluated all of the example pairs based on a multidimensional rubric. Our preliminary results demonstrate that by selecting appropriate features, our data-driven examples can be more relevant than both baselines while retaining the usefulness and interpretability. These promising results suggest that our method can generate adaptive, data-driven examples automatically with quality similar to that of expert-authored examples. Even though our method is based on a block-based programming environment, it is language-agnostic and may also be generalized to textual programming languages; though further studies are needed to verify this proposed generalizability.

Limitations: This preliminary study relied on experts ratings to assess example quality, yielding valuable insight, but future work is needed to determine whether these results translate into improved student performance and learning. For example, even a well-rated example pair may still give away too much of the answer and impede learning. We also do not know how our results will generalize beyond the single assignment evaluated here, since we only applied this method to a short, block-based problem with a solution that is usually 7-10 lines of code. In our DDF-IS condition, we had an expert interactively select the feature to be presented, rather than a student. As noted earlier, this is likely an optimistic implementation, as we do not know whether students can effectively select examples. Finally, all conditions used the SourceCheck algorithm to select similar examples to the student’s code, but other approaches may better capture example relevance (e.g. [HB17]). We are currently planning a student evaluation of the DDF algorithm to address these limitations.
This chapter proposes the Engagement, Cognitive Load, and Extent of Programming (ECLEP) framework, relates it to other empirical studies presented in prior chapters, and makes recommendations based on this framework for the design of instructional supports for programming.

7.1 ECLEP Framework

This section introduces the Engagement, Cognitive Load, and Extent of Programming (ECLEP) framework, which is based on the ICAP framework and Cognitive Load Theory. The ECLEP framework characterizes instructional supports in the domain of programming, based on their cognitive engagement, cognitive load, and level of programming. The framework can be used to guide the design of instructional supports and interpret the empirical results in this work. As reviewed in Section 2.2.3 of Chapter 2, the Interactive-Constructive-Active-Passive (ICAP) framework defines cognitive engagement activities based on students’ apparent behaviors during a learning activity [CW14]. Cognitive Load Theory (CLT) describes the relationship between mental effort and learning, with a focus on implications for the design of instructional tasks. To inform the design of effective
instructional support, I combine CLT and the ICAP framework because they address complementary aspects of learning. CLT suggests designing instructional support to minimize cognitive load, while the ICAP framework recommends incorporating instructional activities with high engagement. Both of these objectives are important, since learning efficiency can be low for instructional supports imposing low cognitive load without high engagement (e.g. BOTS WEs in Chapter 3), or for instructional supports with high engagement level imposing high cognitive load (e.g. problem solving in Snap!). There is also an inherent tension between the suggestions of CLT and the ICAP framework. As Chi et al. argue, a higher level of engagement will be generally associated with higher cognitive load, since each engagement level requires additional and more diverse types of learning activities [CW14]. We therefore cannot always optimize for both high engagement and low cognitive load. Based on CLT, if a learning activity overloads student cognitive capacities, their learning would be impeded. Our framework helps show the potential tradeoffs between engagement and cognitive load, as well as the amount of agency that students have in creating programs, and explains how these factors may influence the outcomes for diverse instructional supports. For example, novices students filling open-ended self-explanation prompts of worked examples (Constructive Engagement Activity [WC14]) may be overwhelmed and thus learn less than students doing scaffolded self-explanation prompts (Active Engagement Activity [WC14]) [Ber09]. Overall, we need to consider and balance both cognitive load and engagement when designing instructional support. Integrating the ICAP framework and CLT would further provide us a tool to interpret the results and guide us to design instructional supports. The ideal instructional support is engaging and would encourage students to actively process the learning material without overloading student cognitive capacity.

In this framework, I categorize the engagement of students in different conditions in my studies using the ICAP framework and then compare the similarities and differences within an ICAP level using CLT. Additionally, I add another element to this categorization, which is specific to computer science, defined as the extent of programming, to describe how much code the students need to write within each instructional support. For any given programming problem, this can be measured by the amount of code that a student needs to write compared with the amount of code that would need to be written to solve the problem from scratch. For example, in the BOTS study presented in Chapter 3, the extent of programming would be none for WEs (no code written), low for buggy WEs (only a few lines of code needed to be written) and high for instructional text (full code written). As Venables et al. pointed out, “the aim of teachers is to produce students who can write programs.” [Ven09]. The ability to write code is important for novices since the programming practice can help enhance student learning and construct schemas (related to building more expert knowledge, as described in Section 2.2.1 of Chapter 2). My hypothesis is that within the same level of engagement and cognitive load, the higher the extent of programming the higher a student’s learning efficiency.

1According to the ICAP framework [CW14], engagement refers to the way that a student engages with the learning material, reflected by the student’s overt behavior during the learning activity.
The extent of programming can be used to further describe the similarities and differences between the instructional supports. Another benefit of identifying the extent of programming within activities is to help instructors choose activities based on their priorities. For example, some CS instructors may place a higher value on activities with more programming.

### 7.2 Study Summaries

To summarize all of the studies present in this work, I situate each result based on the ECLEP framework. The interventions evaluated in this work including BOTS problem solving (PS), BOTS erroneous worked examples (EWEs), BOTS worked examples (WEs), Snap/PS, Snap/incomplete Worked Examples with Self-explanation prompts (IWE+SE), Snap! Parsons Problems (PPs), and Snap! Example-based Feedback (EF). The instructional supports evaluated in this work along with learning outcomes are summarised in Table 7.1. It is important to note that we should be careful when comparing across studies, since they have different factors such as populations, study duration, and learning goals. Additionally, the same label (e.g. Medium cognitive load) in this table may not mean the exact same thing in one study as another. Therefore I carefully discuss this at the end.

To categorize each instructional support, we need to consider all three dimensions including cognitive engagement mode, cognitive load, and extent of programming. The instructional supports can be categorized into one of the four modes based on the ICAP framework and categorized into Low, Medium, and High cognitive load imposed on students based on CLT. They can be further categorized into None, Low, Medium, High, and Full based on the extent of programming.

As mentioned in Section 2.2.3 of Chapter 2, based on the eight assumptions of the ICAP framework, we can classify students cognitive engagement into Passive, Active, Constructive, and Interactive. Generally speaking, most instructional supports can be classified into different engagement modes, depending on how students interact with them. For example, just reading a worked example would be classified into Passive mode; Consciously highlighting different parts of a worked example and summarizing it would be classified into Active mode; Self-explaining a worked example should be categorized into Constructive mode; Discussing a worked example and each others’ corresponding self-explanations with a partner can be considered as in Interactive mode. Based on the variation of students’ interactions with the instructional support, we should design instructional support that encourages students to remain in higher engagement mode. It is likely that different students may engage in different modes of engagement within the same instructional support, such as some students just reading a worked example while some highlight different parts of the worked example. In this case, Chi et al. suggest to classify the mode of engagement based on how the majority of students interact with it (i.e. the greater probability assumption [CW14]). For example, when reading a textbook, if the majority of students are underlining or highlighting important concepts or examples for most of the time, then it is likely that they are engaging actively instead of passively (i.e.
just reading the book without doing anything). In the ICAP column of Table 7.1, I categorized each instructional support in this study based on the most common engagement mode observed in the data, as described in detail later. It is worth noting that this chapter engages in a post hoc analysis to apply the ECLEP framework to my data. I was therefore unable to categorize students’ mode of engagement by observing their overt behaviors during the studies, as is suggested by the ICAP framework. Instead, I analyzed students’ artifacts that they created during the practice activities (i.e. training phases). This is supported by the Analyzing the outputs assumption of the ICAP framework that we need to analyze students’ output to confirm their actual mode of engagement. In my studies, I classify students’ mode of engagement during the practice phase as follows: Passive if students just read the instructions without paying much attention to it; Active if students applied what they learned to solve the problem; and Constructive if they are able to apply the new information from instructions to finish the problem. None of my studies ask students to work in pairs, so students’ mode of engagement are unlikely to be Interactive. Remember that higher engagement activities may still overload students’ cognitive capacity and impede learning, so we need to consider the cognitive load imposed by an instructional support after categorizing the engagement mode.

Based on the Cognitive Load Theory (described in Section 2.2.1 of Chapter 2), students’ total cognitive load comes from the combination of intrinsic load and extraneous load, we can classify the imposed cognitive load based on the addition of intrinsic load and extraneous load imposed by the learning material. An effective instructional support should not overload students’ cognitive capacity. In the CLT column of Table 7.1, the cognitive load of different instructional supports are categorized from Low to High. For example, novices solving problems often use means-ends analysis which would impose heavy cognitive load on students; therefore, problem solving has the highest cognitive load among all the instructional supports. According to CLT, studying worked examples would impose lower cognitive load than problem solving, thus worked examples have low cognitive load compared to problem solving. The rationale for the categorization of each instructional support is discussed later. However, it is important to note that since we did not directly measure cognitive load in most studies, the reported level of cognitive load is hypothesized based on the design of the instructional support and other results from the study (e.g. time on task, learning). In this work, I classify the cognitive load based on the time students spent during the training phase\(^2\) if we did not measure their cognitive load. The time students spent during learning can be seen as an indirect cognitive load measurement, as pointed out by Brünken et al., “learners’ time-on-task can be seen as an indicator for different load levels.” [Brü03] However, we need to be cautious when using the time-on-task, since it only indirectly linked to cognitive load and can be affected by many other factors [Brü03].

\(^2\)I did not apply this method for Study 2, but used a cognitive load survey results instead, since we controlled the time for the Study 2.
the BOTS problem solving condition (Text condition in Chapter 3, since students solve the puzzles by writing code from scratch, the extent of programming is High. Again, our goal is to design effective instructional supports. My hypothesis states that within the same engagement mode and cognitive load, we can use the extent of programming to further differentiate the instructional supports and predict their effectiveness.

**Table 7.1 Instructional supports categorization based on ECLEP framework.**

<table>
<thead>
<tr>
<th>Study</th>
<th>ICAP</th>
<th>CLT</th>
<th>Extent of Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOTS PS</td>
<td>Active</td>
<td>High</td>
<td>Full</td>
</tr>
<tr>
<td>BOTS EWEx</td>
<td>Constructive</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>BOTS WEx</td>
<td>Passive</td>
<td>Low</td>
<td>None</td>
</tr>
<tr>
<td>Snap/PS</td>
<td>Constructive</td>
<td>High</td>
<td>Full</td>
</tr>
<tr>
<td>Snap/IWE+SEx</td>
<td>Constructive</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Study 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-lab Snap/PS</td>
<td>Constructive</td>
<td>High</td>
<td>Full</td>
</tr>
<tr>
<td>Snap/PPx</td>
<td>Constructive</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Study 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Snap/EFx (predicted)</td>
<td>Interactive/Constructive/Active</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

* Instructional support conditions had better learning outcomes than others.
+ Instructional support conditions were completed faster.

The following sections present the rationale of the categorization of each study and interpretation of study results based on the ECLEP framework. As mentioned above, according to the ICAP framework, the engagement mode of an instructional support depends on how students behave, as indicated by the intended versus enacted and analyzing the outputs assumptions (described in Section 2.2.3 of Chapter 2). The instructional supports in this thesis are unlikely to be categorized into Interactive, since the students were programming alone during studies. For the ICAP categorization of instructional supports, I will explain the most common mode of engagement suggested by data.

### 7.2.1 Study 1: Instructional Text, Worked Examples, and Erroneous Worked Examples in BOTS

This section describes the rationales of the categorization of each instructional support in this study and discusses the implications of these categorizations for interpreting the studies' results.
7.2.1.1 BOTS Problem Solving (PS)

**Active Engagement Mode:** I classify the engagement mode for BOTS-PS as Active. The Active mode indicates that students apply what they have previously learned to solve similar problems. In other words, the majority of students solve the problem without applying new knowledge introduced in the instructions. According to the Section 3.3 in Chapter 3, the majority of students doing problem solving (i.e. Text condition) did not follow the instructions and did not apply advanced commands (i.e. loops) to solve the puzzle. Most of them just used basic commands to solve the experimental puzzles, suggesting that they were only applying existing knowledge. The engagement mode is unlikely to be Passive since students need to interact with the environment extensively to solve the puzzle. Specifically, students may digest the textual explanations passively, but they still need to solve the puzzle by writing code.

**High Cognitive Load:** According to Cognitive Load Theory, problem solving would impose heavy cognitive load to students since they would be likely to use means-ends analysis to solve the problem (i.e. students need to save the intermediate goal in their working memory and meanwhile search space to move to the next step until solving the problem). This is supported by the results in Section 3.3 of Chapter 3 that most students just used basic commands to solve the experimental puzzles instead of more advanced commands such as loops or functions. Additionally, they spent the longest time among all three conditions on average to solve the experimental puzzles. Therefore, I classify problem solving as imposing a high cognitive load on students. Specifically, problem solving from scratch offers little support to reduce cognitive load for students, especially novices.

**Full Extent of Programming:** Since students need to write code from scratch to solve the puzzles, the extent of programming is full.

7.2.1.2 BOTS Erroneous Worked Examples (EWEs)

**Constructive Engagement Activity:** I categorize the engagement mode for BOTS-EWEs as Constructive. The Constructive mode means that students not only applied what they already knew but also integrated new information to solve problems. Specifically, the most students should reflect on the buggy code, and focus on finding the errors and fixing them instead of deleting all the code and constructing a solution using basic commands from scratch. Based on the results in Section 3.3 in Chapter 3, the majority of students did finish the experimental puzzles properly, by correctly finding and fixing bugs. Therefore, the engagement mode of students in Bugs condition is Constructive. Similar to BOTS problem solving, fixing erroneous WEs is unlikely to be Passive since students need to find and fix bugs, or delete the code and create their own solution to solve the puzzle.

**Medium Cognitive Load:** The EWEs already include good examples for using advanced commands and therefore helped reduce the search space that students needed to explore. Students instead needed to focus on understanding the buggy code, finding and fixing errors instead. Even
though there are only one or two bugs inside the example code, the result in Section 3.3 in Chapter 3 shows that the time EWEs students spent on experimental puzzles is nearly four times longer than the time spent by WEs students, indicating that it may impose heavier cognitive load to them than worked examples. Additionally, EWEs students took less time on average than PS students, so it is likely that the imposed cognitive load is lower than problem solving. Hence, I classify EWEs would impose medium cognitive load on students.

**Low Extent of Programming:** Students only need to find one or two errors in the code and fix them, which does not require students to write much code. The results in Chapter 3 suggest that most students (4/5) did find and fix errors instead of deleting all the code and write a solution from scratch. Therefore, the extent of programming for Bugs condition is Low.

7.2.1.3 **BOTS Worked Examples (WEs)**

**Passive Engagement Activity:** I classify the engagement mode for BOTS-WEs as Passive. The Passive mode indicates that students just read the instructions and view the example code without paying much attention to them. The data show that the majority of students finish the WE puzzles without correctly answering the questions after each puzzle. According to Section 3.3 of Chapter 3, the majority of students only ran the example code once and did not answer the question correctly, indicating they probably skipped the experimental puzzles without paying attention to the example code. Therefore, students studying BOTS WEs are likely to be in the Passive mode.

**Low Cognitive Load:** As with the EWEs, the WEs would impose low cognitive load to students, and it would be even lower than EWEs, since students do not need to debug the example code. This is supported by the results in Section 3.3 of Chapter 3 that Examples (WE) students spent significantly less time than Bugs (EWE) students and Text (PS) students. Therefore, compared with other instructional supports, the BOTS WEs impose low cognitive load.

**None Extent of Programming:** Since students do not need to write any code in BOTS WEs, the extent of programming is None.

We can situate the results of this study in the ECLEP framework. According to the ICAP framework, the learning outcome (i.e. students’ performance on solving the assessment puzzles in this case) would follow this order: erroneous worked examples (EWEs; **Constructive**) > problem solving (PS; **Active**) > worked examples (WEs; **Passive**). Our results are partially consistent with ICAP’s predictions. Specifically, according to the Section 3.3 of Chapter 3, students’ performance on the post-test puzzles indicate that EWEs > PS = WEs. However, we did not find evidence suggest that PS > WEs but instead found WEs > PS. According to Section 3.3 of Chapter 3, WEs students spent much less time than PS students on the training phase but they performed similarly on post-test. This finding is predicted by CLT, since WEs impose a lower cognitive load than PS. However, both

---

3Remember that students did not demonstrate the new knowledge they learned while solving the experimental puzzles.
WE students and PS students did not learn much, as evidenced by the fact that they did not apply the advanced commands to solve the assessment puzzles properly. This result can be explained by the ECLEP framework, which suggests that both high cognitive engagement and low cognitive load alone are important for learning. According to CLT, it is likely that the problem solving imposed an overly heavy load to students so they did not learn well. Therefore, even though PS had a higher level of engagement, it probably overloaded students’ cognitive capacity and thus lead to poor learning. Similarly, even though WEs would impose low cognitive load to students, it did not positively impact their outcomes since they were in Passive mode. Based on these results, my hypothesis for an effective instructional support is that the engagement level should be at least Constructive without overloading student cognitive capacity. In this study, I believe that the different outcomes can be explained using the dimensions of ICAP and CLT, and therefore do not consider the extent of programming in predicting learning outcomes.

7.2.2 Study 2: Snap! Incomplete Worked Examples With Self-Explanation Prompts

7.2.2.1 Snap! Problem Solving (PS)

Constructive Engagement Activity: I categorize the engagement mode for Snap!PS as Constructive. The Constructive mode indicates that students can integrate their prior knowledge with new information learned while using the instructional support (instructional text in this case) and apply the new knowledge to solve problems. According to Section 4.3 of Chapter 4, the majority of students (80% for problem 1 and 60% for problem 2) in the problem solving group did follow the instructions and finish the WE objectives of both problems, which require students to apply concepts such as conditionals, loops, and procedures. Therefore, students’ mode of engagement are likely to be Constructive.

High Cognitive Load: According to CLT, problem solving would impose the highest cognitive load to students since they were likely to use mean-ends analysis to solve the puzzles. In this study, we measured students’ cognitive load using the CS Cognitive Load Component Survey (CS CLCS) [Mor14]. The cognitive load survey results suggest the problem solving imposes heavier cognitive load than the incomplete WEs for problem 2 but similar load for problem 1. This can be explained by the element interactivity effect and will be discussed later when I classify Snap! IWE+SE.

Full Extent of Programming: Students need to write code from scratch to solve the problem.

7.2.2.2 Snap! Incomplete Worked Examples with Self-explanation Prompts (IWE+SE)

Constructive Engagement Activity: I classify the engagement mode for Snap! IWE+SE as Constructive. According to Section 4.3 of Chapter 4, all the students in IWE+SE condition finished the worked example steps, and some students (100% for problem 1 and 44% for problem 2) finished at least one more objective after finishing the worked example steps. Moreover, the results show that students
who did IWE+SE finished more objectives on average than the students who wrote code from scratch. Finishing extra objectives indicates that students were able to transfer their knowledge from the worked example steps. Hence, IWE+SE students are likely to be in Constructive mode.

**Medium Cognitive Load:** According to CLT, WEs would impose relatively low cognitive load to students since it helps reduce students’ search space. This design also requires students to write some code after studying worked examples, therefore IWE+SE would impose heavier cognitive load to students than WEs but less cognitive load than problem solving. The cognitive load measurement survey results suggest that the worked examples imposed similar cognitive load to students for problem 1 but imposed much less cognitive load to students for problem 2. This can be explained by the the element interactivity effect (described in Section 2.2.1 of Chapter 2), that problem 1 may be be simple enough to solve with the help of instructions so that the predicted improved learning efficiency of worked examples may not be realized. Problem 2 is designed to be more difficult than problem 1, students are likely to benefit more from the IWE+SE. As the results show, IWE+SE students they accomplish more objectives on average than students writing code, though the difference is not significant. Therefore, compared to Snap! problem solving, the Snap! IWE+SE support imposes medium cognitive load.

**Medium Extent of Programming:** Nearly half core objectives of the problem are done by the worked example steps, so students only need to finish the rest of the problem instead of a full problem.

Based on the ICAP framework, the learning outcome of students who did Incomplete WEs with self-explanation prompts (Constructive) should be similar to students who did problem solving (Constructive). Our results are in-line with the prediction that we should not observe significant difference between the two groups on the two conceptual post-tests. Additionally, the majority of students in both groups finished the core objectives of the problems. According to CLT, students should benefit more from learning from IWE+SE. For the in-problem performance, if we limited the time (e.g. 20 minutes), we would also see significant differences between the two groups on the amount of finished objectives. The results in Section 4.3 of Chapter 4 show that the incomplete WEs with self-explanation prompts save students at least 42% of time and help them to finish all the WE objectives. Overall, students with incomplete WEs with self-explanation prompts finished more objectives than students who wrote code to solve the problem from scratch, although this difference is not significant. In summary, the lack of significant difference between conditions may be due to the fact that students are in the same mode of engagement, despite some differences in cognitive load.
7.2.3 Study 3: Snap/Parsons Problems

7.2.3.1 In-lab Snap/Problem Solving (PS)

**Constructive Engagement Activity:** I categorize the engagement mode for In-lab Snap/PS as Constructive. The Constructive mode indicates that students can apply their knowledge to solve novel problems or integrate prior knowledge and new information to form new knowledge. The in-lab problems are novel to the students, and common solutions of those problems involve new concepts and blocks that were introduced by the teaching assistants during labs. Students are expected to solve those in-lab problems by using a combination of those new blocks and blocks that they already knew. According to Section 5.3 of Chapter 5, the majority of students in the problem solving condition performed well for those assignments, and they did apply new blocks to solve lab assignments, as indicated by their class grades, since the grading rubrics capture the new blocks that students need to use in lab assignments. Therefore, the majority of students were likely to be in the Constructive mode of engagement.

**High Cognitive Load:** According to CLT, problem solving would impose the highest cognitive load to students since they were likely to use mean-ends analysis to solve the puzzles. Though we did not analyze how students solve the problems, the results in Section 5.3 of Chapter 5 suggest that students spend 2-3 times longer on average to solve the in-lab assignments than students solving Parsons problems. Therefore, I classify problem solving would impose high cognitive load on students.

**Full Extent of Programming:** Students need to write code from scratch to solve the problem.

7.2.3.2 Snap/Parsons Problems (PPs)

**Constructive Engagement Activity:** I classify the engagement mode for Parsons Problems (PPs) as Constructive. Similar to Snap/PS, students who solve PPs were able to use the new, advanced blocks to finish in-lab assignments as expected. According to Section 5.3 of Chapter 5, the majority of students use all the available blocks to solve the problem correctly and it is likely that they formed new knowledge by solving the Parsons problems. Therefore, they are likely to be in the Constructive mode while doing Parsons problems.

**Medium Cognitive Load:** The limited available blocks in a Parson Problem help reduce the search space the students must navigate. Therefore, this type of support would impose less load than problem solving, as indicated by the result that students spent much less time on finishing the Parsons problems than writing code. However, students still need to construct a solution by putting

---

4 Since we did not measure students’ cognitive load in this study, their time spent on the in-lab assignments was used to estimate the cognitive load.

5 The rationale for the categorization is similar to that in Snap/PS.
all the available blocks together, so it is likely that PPs would impose heavier load to students than WEs. Therefore, PPs would impose medium cognitive load to students.

**High Extent of Programming:** In Parsons Problems, students need to rearrange code to solve the problem instead of writing code from scratch. For the traditional problem solving in a block-based programming environment, students need to identify, select, and use the blocks from a list of commands from different categories. In my design of block-based Parsons problems, the extent of programming is close to problem solving since students not only need to rearrange existing blocks, but also to place the parameters and variables correctly. However, even with this design, students do not need to create variables, parameters or custom blocks, which are needed in problem solving. Therefore, Snap/PPs have lower extent of programming than problem solving.

According to the ICAP framework, the learning outcomes of block-based Parsons Problems (Constructive) are similar to In-lab Snap/Problem Solving (Constructive). According to CLT, students are likely to benefit more through solving Parsons problems than problem solving. Compared to problem solving, Parsons problems are likely to impose less cognitive load to students due to the limited available blocks. This is supported by our data that students solving Parsons problems finished the lab assignments much quicker than students writing code from scratch, and they performed similarly on the following homework assignments. Overall, based on the categorization of different instructional supports in Snap/, the learning outcomes would be predicted by the ECLEP framework as Snap/Parsons Problems > Snap/Problem solving, which is supported by our data. Similar to Study 1, I do not consider the extent of programming in predicting learning outcomes of Snap/PPs and in-lab Snap/PS, since the different outcomes can be explained using the dimensions of ICAP and CLT.

### 7.2.4 Study 4: Snap/Example-based Feedback

#### 7.2.4.1 Snap/Example-based Feedback (EF)

In this section, I propose categorizations for Example-base Feedback, but since we do not have student data yet, they are only predictions. The modes of Snap/Example-based Feedback can be Interactive, Constructive, or Active, depending on how students interact with the system. It is unlikely to be Passive since students need to write code to solve the problems. Below, I discuss the conditions under which EF could be Interactive, Constructive and Active:

- **Active:** If the students abuse the help or avoid seeking help when they need it, they just use their existing knowledge to solve the problem and ignore the new knowledge components embedded in the problem.

- **Constructive:** Students who seek help properly when they need it are likely to form new knowledge by learning from the example code. They would self explain the example code and...
apply the new knowledge to their code.

- **Interactive:** Students are studying in pairs and both are already in Constructive mode would be in Interactive mode if they discuss their thoughts about their strategies and solutions for the problem. They would form new ideas by talking with each other.

**Medium Cognitive Load:** The imposed cognitive load is dependent on how students use the support. If students only use the support when they need it, it would impose higher cognitive load than worked examples (since it requires students to apply the example code to their own solutions), but lower cognitive load than problem solving (since example-based feedback can help reduce the search space by showing example code to accomplish a meaningful feature). I therefore categorize it as Medium cognitive load. However, if students do not use the help at all when they actually need it, this would be the same as a normal problem solving and impose high cognitive load to students. In another case, if students abuse the help and just copy the suggested code, this is likely to be treated as worked examples and thus imposes a lower cognitive code to students.

**High Extent of Programming:** Students need to write code from scratch. Even with the support, they still need to apply what they learned from the example-based feedback to their solution.

Example-based feedback is adaptive and is suitable for learners with different levels of programming knowledge. It strikes a balance between WEs and PS. The activity engagement mode is at least Active and the support would help reduce cognitive load, therefore students would have more mental resources to digest the learning material and thus improve learning. Additionally, example-based feedback brings benefits of worked examples into problem solving and students are likely to benefit from it when instructors prefer to do problem solving.

### 7.3 Summary & Instructional Support Recommendations

My hypothesis for the ECLEP framework is that **given the same mode of engagement and cognitive load, the higher the extent of programming the higher a student's learning efficiency will be.** Based on the study categorizations in Table 7.1, there are two instructional supports with the same level of engagement and cognitive load: Snap/IWE+SE and Snap/PPs. Based on the ICAP framework and CLT, we cannot differentiate Snap/IWE+SE and Snap/PPs, since both instructional supports are Constructive and impose Medium cognitive load. And we are not able to predict which one is more effective based on either the ICAP framework or CLT. However, by applying the ECLEP framework, we are able to compare Snap/IWE+SE and Snap/PPs and predict their effectiveness. According to the hypothesis, the predicted learning outcomes would be Snap/IWE+SE less than that for Snap/PPs, because Snap/PPs has higher extent of programming than Snap/IWE+SE. Though I did not conduct experiments to compare those two instructional supports, the results from Study 2 and Study 3 may suggest that Snap/PPs have stronger impact than Snap/IWE+SE. They were both compared with
Snap/PS, but we did not find significant difference between Snap/IWE+SE and Snap/PS. However, we did find significant difference between Snap/PPs with Snap/PS, with a large effect size. This may indicate that Snap/PPs may have a larger impact than Snap/IWE+SE. However, we cannot compare the results directly since the population and problems were different. While my current data provides only weak evidence for this hypothesis, this example shows how the ECLEP framework is useful to further differentiate the instructional supports. Testing the hypothesis of the ECLEP framework can be a future direction of this work.

Based on the hypothesis and my study results, I would recommend teaching novices by solving Parsons problems since it is engaging, has a high extent of programming, and is unlikely to overload students’ cognitive capacity. Novices are unlikely to learn efficiently if they do problem solving or just explain worked examples. If students have different levels of prior knowledge, I would recommend using the example-based feedback since it is adaptive and hopefully students with high prior knowledge would benefit from it as well. Additionally, some instructors may prefer full extent of programming, and thus may continue to require novices to solve almost all of their homework problems from scratch. Therefore, example-based feedback may be a good adaptive option for such classes or instructors.
This chapter revisits the research questions, presents the contributions, and points out directions for future work about instructional supports in programming.

8.1 Research Questions

My research aims to develop insights into designing effective, theoretically-grounded instructional supports in novice programming environments. In this section, I discuss how this work has addressed the following three high-level research questions:

- **RQ1**: How do different types of worked examples compare to problem solving in novice programming environments?
- **RQ2**: How do Parsons problems compare to problem solving in novice programming environments?
- **RQ3**: How can we use data to provide high-quality example-based feedback automatically?

8.1.1 Research Question 1

RQ1: *How do different types of worked examples compare to problem solving in novice programming environments?* In the context of this research question, I studied worked examples, erroneous
worked examples, and incomplete worked examples with self-explanation prompts. This research question has been addressed in Chapters 3 and 4. Chapter 3 discusses the design and evaluation of worked examples and erroneous worked examples in a block-based programming game named BOTS. Chapter 4 introduces the design and evaluation of incomplete worked examples with self-explanation prompts in a block-based programming environment called Snap!. By comparing different types of worked examples with writing code from scratch, I found the erroneous worked examples and incomplete worked examples with self-explanation prompts are promising to help novices learning programming. However, there are large challenges to implementing effective worked examples for novice programming, and that students may not actively engage in learning from worked examples.

Specifically, I found that both the erroneous worked examples and incomplete worked examples with self-explanation prompts are promising instructional supports to help novices learn programming. As shown in Section 3.3 of Chapter 3, students who studied erroneous worked examples in BOTS finished the performance posttest puzzles quicker and had fewer low-performing students than students who studied instructions to write code from scratch. However, the students who studies worked examples suggest that they may not attend to worked examples. By incorporating self-explanation prompts and adding practice objectives in worked examples, in Study 2, we found that students accomplished more objectives in the first half of problem solving activities than students writing code (as shown in Section 4.3 of Chapter 4). This suggests that incomplete worked examples with self-explanation prompts have the potential to improve students’ learning efficiency. Study 2 informs that novices benefit from finishing the incomplete worked examples, which combines worked examples and problem solving. This is in line with findings from Study 1 that students learn better from erroneous worked examples, which also comprise the elements of worked examples and problem solving.

Overall, by reflecting on the results of both Study 1 and Study 2, my results suggest that implementing effective programming worked examples is challenging and identified that there is a tradeoff between engagement, learning, and time spent on the learning activity. Worked examples are likely to save students time, but students may not actively engage in learning from them. Students may benefit more from learning from erroneous worked examples or incomplete worked examples with self-explanation prompts. Interleaving worked examples and problem solving has also been shown to be effective in teaching students programming [TR93]. In Chapter 7, I discuss how to use a theoretical framework to inform the design of instructional support that balances engagement, learning, and time spent.
8.1.2 Research Question 2

**RQ2: How do Parsons problems compare to problem solving in novice programming environments?**

The second research question has been addressed in Chapter 5, which presents the design and evaluation of Parsons problems in Snap!. I designed novel, block-based Parson problems and evaluated their impact using a quasi-experimental design in a classroom setting. The major finding of this research is that the block-based Parsons problems saved students 15 to 17 minutes (43.6% to 65.2% of total time) on 30-minute lab assignments and they performed at least as well on subsequent assignments compared to writing code. This indicates that Parsons problems can improve student learning efficiency, which is consistent with previous studies [Eri17; Eri18; Har15]. This is likely due to a combination of the fact that Parsons problems dramatically reduce the programming solution space, and our findings that they prevented unproductive programming behaviors such as devising a solution, searching for needed components, and composing them together. The positive impact of Parsons problems on student learning efficiency shows a promising direction for programming instructional supports that balance the benefits of lower cognitive load and reduced time for worked examples and the effort and engagement needed for problem solving.

8.1.3 Research Question 3

**RQ3: How can we use data to automatically provide high-quality example-based feedback in novice programming environments?**

The third research question has been addressed in Chapter 6, which introduces an algorithm to generate adaptive, data-driven example-based feedback from student data. I asked experts to rate the quality of example-based feedback that generated from different approaches using a rubric (shown in Appendix). The results presented in Section 6.3 of Chapter 6 suggest that the data-driven approach could be used to generate adaptive, example-based feedback that are more relevant than example-based feedback derived naively from student data. Additionally, the example quality can be improved further by leveraging the interactivity with students.

Unlike other studies in this work, one major limitation of Study 4 is that the example-based feedback evaluation is based on experts’ ratings, rather than student learning. This evaluation was intentionally chosen in order to test the feasibility of my method before building the whole system and conducting a user study. The results of Study 4 show that this method is suitable for generating relevant example-based feedback with similar quality of expert-authored ones. In the near future, I would like to work with others to build the whole system and evaluate the example-based feedback with students.

8.2 Contributions

This dissertation offers the following primary research contributions:
1. The design and evaluation of instructional text, worked examples, and erroneous worked examples in a novice programming environment (see Chapter 3). The worked examples did save students time but they did not actively engage in learning from the worked examples. Similarly, students often skipped the instructional text and therefore did not learn well. The erroneous worked examples may be a promising strategy to teach novices programming. From this study, I identified the tradeoffs in the design of instructional supports in terms of engagement, learning, and time spent on the learning activities.

2. The evaluation and design of incomplete worked example with self-explanation prompts in a novice programming environment. While the design and evaluation are in the context of Snap!, the results are likely to be generalizable to other block-based programming environments. This study shows a promising direction for balancing the reduced cognitive load of worked examples with the increased active engagement of problem solving in instructional support design.

3. A novel design of Parsons problems in a novice programming environment. The design of block-based Parsons problems is introduced in Section 5.2.1 of Chapter 5. This novel design allows students to interact with three different modes of inputs including editable, editable with a default value, and fixed.

4. A comparison of the impact of block-based Parsons problems and problem solving on student learning efficiency. The results of Study 4 showed that Parsons problems can improve student learning efficiency maybe because Parsons problems can dramatically reduce the programming solution space and prevent unproductive behaviors.

5. A novel method for generating data-driven example-based feedback, which enables adaptive example-based feedback for students. This study demonstrates the potential for providing relevant, adaptive and high-quality example-based feedback to students.

6. A new framework that integrates the ICAP activity engagement framework with Cognitive Load Theory to explain the differences among instructional supports based on how well they promote engagement and effort while reducing cognitive load.

8.3 Future Work

This work provided insights into the design and evaluation different instructional support in novice programming environments and used data to generate the instructional support automatically. The next step of this work is to evaluate the adaptive, data-driven example-based feedback with novice programmers. While my research demonstrates that it is possible to develop effective instructional
supports in block-based programming environment, practitioners should balance engagement, learning, and time spent while designing the instructional supports. Future directions of this work include validating the ECLEP framework, developing algorithms to automatically generate data-driven instructional supports, and providing adaptive instructional supports to students.

One direction of this work is to validate the ECLEP framework. Since the ECLEP framework has three dimensions (i.e. ICAP, CLT, Extent of Programming), we can design experiments to validate the framework by controlling the level of engagement and cognitive load and varying the extent of programming. For example, to test the hypothesis that within the same engagement and cognitive load, the higher the **extent of programming** the better for students, we can conduct experiments to compare Snap/Parsons problems and Snap/Incomplete Worked Examples with Self-explanation prompts. This hypothesis would be supported if students solving Snap/Parsons problems performed better than students solving Snap/Incomplete Worked Examples with Self-explanation prompts. The ECLEP framework predicts that the maximum learning efficiency would come from an instructional support that requires Interactive engagement, with high extent of programming and without overloading students' cognitive capacity. Based on this prediction, an example of ideal instructional support could be asking students to do pair-programming or peer-instruction while solving Parsons problems. By doing this, students are more likely to increase the level of engagement to Interactive mode and thus improving the learning. Additionally, this is suggested by the CLT collective working memory effect, that students would learn better when studying collaboratively than studying alone.

For the data-driven instructional supports generation, we can develop methods to convert worked examples or data-driven example-based feedback into Parsons problems. For example, based on a correct student solution, we could convert it to a Parsons problem and ask other students to solve. Additionally, we could also generate Parsons problems based on each feature of the example-based feedback (described in Section 6.2.2.1 of Chapter 6) and then ask students to solve those Parsons problems.

This work focuses on the impacts of instructional supports on novices. However, we usually have students with different levels of prior knowledge in reality. In Study 2, where we evaluated the incomplete worked examples along with self-explanation prompts, the results also indicate that some of the participants had high prior knowledge and did not appreciate the instructional support. To further improve students learning, we could provide adaptive instructional supports to students. By assessing students knowledge, we can select corresponding instructional support to them. For example, if a student does not have any programming experience, we could provide worked examples with self-explanation prompts or Parsons problems. If a student already knows the concepts, we can provide problems with example-based feedback so the students can ask for help only when they need it.
BIBLIOGRAPHY


[Hic14] Hicks, A. et al. “Part of the game: Changing level creation to identify and filter low quality user-generated levels”. FDG. 2014.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title and Authors</th>
</tr>
</thead>
</table>


APPENDIX

A

ASSIGNMENTS

A.1 Incomplete Worked Example with Self-explanation Prompts Study Assignments

A.1.1 Daisy Design

A.1.1.1 Project description

In this project, you will write code to draw the geometric design shown below, explore its variations and use it draw more complex designs like those below (shown in the following Table).

Turn on the ‘Turbo mode’ if you want the sprite to draw things faster:
<table>
<thead>
<tr>
<th>Draw Daisy</th>
<th>Draw Colorful Daisy</th>
<th>Two Daisies</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td>Three Daisies</td>
<td>Triangle Daisies</td>
<td>Hexagon of Daisies</td>
</tr>
<tr>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
</tbody>
</table>
A.1.1.2 Instructions

1. Create a custom block to draw a circle
   - The entire design is made up of circles. So the first thing to do is to write code that will create a circle. To do this, create a custom block via the menu option.
   - Hint: A regular polygon with 30 or more sides is a good approximation for a circle.

2. Make a custom “Draw Daisy” block to draw the daisy
   - Now that you can draw a circle, you can generate the daisy design by rotating your sprite a bit at the end of each circle drawn.
   - In the design above there are 24 circles. How many degrees must the sprite turn each time a circle is drawn in order to make a full cycle of 360° when all the circles are drawn?
   - Create a custom block to do this.

3. Draw Colorful Daisy
   - Change the color while drawing each circle in Daisy.
   - Create a custom block to do this.

4. Draw different designs
   - Make your own variation on the daisy design
     (a) Draw two daisies
     (b) Draw Three daisies
     (c) Draw Triangle daisies
     (d) Draw Hexagon of Daisies
   - Add more code to draw different daisy designs!
     (a) Expand your code to draw customized daisy designs

Remember to save and submit your program when you finished!!

A.1.2 Spiral Polygon

A.1.2.1 Project description

In this project, you will build blocks that make your sprite draw squiral-like shapes like those below.
<table>
<thead>
<tr>
<th>Squiral</th>
<th>Colorful Squiral</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Squiral Image" /></td>
<td><img src="image2.png" alt="Colorful Squiral Image" /></td>
</tr>
<tr>
<td>Give the block an input that allows the user to set the number of times the sprite completes a full rotation around the center. (The picture has 5 rotations, i.e. each side has rotated 5 times).</td>
<td>Change the color while drawing each side.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spiral Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Spiral Polygon Image" /></td>
</tr>
<tr>
<td>Create a new block to draw squiral-like polygon. Your custom block have two inputs that allow the user to set the number of times the sprite completes a full rotation around the center, and the side of the polygon.</td>
</tr>
</tbody>
</table>
A.1.2.2 Instructions

1. Create custom block to draw a squiral
   • The entire squiral design is made up of squares but with increasing side length. So the first thing to do is to write code to draw a square. And then increase the side length while drawing the square by using a variable.
   • Hint: You can create a variable by clicking on tab. And you may use to set and to change the value of your variable.
   • Afterwards, create a custom block (The input, or blank, is for the number of rotations) to draw the squiral, and ask the user how many rotations to make. Be sure to store this number in a variable. A full square (has 4 sides) represents one rotation, so if a user specifies 4 rotations, you need to draw 16 sides.

2. Create custom block to draw colorful Squiral
   • Now that you can draw a squiral, you can change the color while drawing each side in squiral. Create a custom block to do this. (The input, or blank, is for the number of rotations)

3. Draw Spiral Polygon
   • Now that you can draw a squiral, create a custom block to draw squiral with different polygon shapes. Your custom block should have two inputs and allow the user to input the side of the polygon and the number of rotations of the polygon. Think about how to extend your current code in to draw polygon and make it squiral-like.
   • Hint: The turning degree for a polygon is \( \frac{360}{\text{sides}} \). For example, when you draw a triangle, each turn is \( \frac{360}{3}=120 \) degrees. When you draw a square, each turn is \( \frac{360}{4}=90 \) degrees. When you draw a hexagon, each turn is \( \frac{360}{6}=60 \) degrees.

4. Draw different designs based on the project description
   • If you have more time, you can create your customized shape and try to make it squiral-like! For example:
A.1.3  Brick Wall

A.1.3.1  Project description

In this project, you will draw a brick wall like this:

Afterwards you can draw a brick wall with your own designs, for example:

A.1.3.2  Instructions

1. **Create a custom block to draw a brick**
   - The first step of building a brick wall is to create a brick. Create a custom block to draw a brick.
   - **Hint:** set the pen size to 10 and move 30 steps to draw a simple brick.
   - Do not forget to set flat line ends to true.

2. **Create custom blocks to draw different rows**
   - There are two different rows in the brick wall. Each row has the same length. Create a custom block to draw the two different rows. Each row has 6 bricks.
3. **Draw brick wall**

- Now you can draw two different rows, create a custom block to draw a brick wall. Hint: You can check the row number to decide whether to use \( \text{rowA} \) or \( \text{rowB} \). For all odd rows, draw \( \text{rowA} \), while for even rows, draw \( \text{rowB} \).
- **Hint**: The \( \text{mod} 2 \) block can help you detect whether the current row is even or odd. For example, if \( \text{currentRow mod 2} \) equals to 0, then the currentRow is even.

4. **Draw brick wall with different designs**

- Now you can create your own shape to draw customized brick wall! Go to the Costumes tab in Snap! and then click \( \text{Paint a new costume} \). Now draw whatever the shape you like. For example:

![Costume interface](image)

- You may draw as large as you like, but make sure to shrink it whenever you are finished in order to fit all the bricks on the screen. Then make sure that your sprite is wearing the “costume”.
- **Hint**: You should duplicate the created costume twice, then edit them to make two partial ends of the "brick".
- To draw a brick, you can use the \( \text{stamp} \) block and move certain steps to draw a brick. When you draw different rows, make sure you change the costume to draw correct length and shape.

### A.2 iSnap Assignments

The iSnap assignment is used in Chapters 5 and 6.
A.2.1 Polygon Maker

Step 1: Create a new block that can draw any polygon. It should have a total of 3 input areas for:

- the size of each side
- the number of sides
- how thick the pen is

Step 2: Perform these tests:

1. # of sides = 4, size = length of 100, thickness = 5
   - This should draw a square
2. # of sides = 10, size = length of 50, thickness = 6
   - This should draw a decagon
3. # of sides = 50, size = length of 7, thickness = 2
   - What do you think this will draw?

A.2.2 Squiral

In this activity you will build a block, in SNAP, that makes your sprite draw a squiral like the one below.

Give the block an argument that allows the user to set the number of times the sprite completes a full rotation around the center. (The picture has at least 5 rotations).
A.2.3  Guessing Game 1

Item 1: The computer chooses a random number between 1 and 10 and continuously asks the user to guess the number until they guess correctly. Item 2: Make sure that your program contains the following:

- Welcome the user to the game
- Ask the user's name
- Welcome the user by name
- Tell the user if their guess was too high, too low, or correct

A.2.4  Guessing Game 2

Please make sure that all items from Guessing Game 1 are included in this assignment, as well as the following:

1. Allow the user to choose minimum and maximum numbers that the random number will be between
2. Keep track of how many guesses the user used
3. Report the number of guesses when the user wins

A.2.5  Pong 1

In 1 player pong, the player will be able to control the paddle with your mouse. The goal is to collect points when the ball bounces off your paddle. The ball will bounce off of all walls (except the wall behind the paddle) and the paddle. You will add the following functionality to the program:

- The mouse will control the paddle moving up and down. Make sure that the paddle doesn't move off the screen.
- When the space key is pressed, the ball will start to move.
- If the ball is on the edge, it will bounce.
- If the ball touches the paddle, it will bounce off of the paddle in the opposite direction.
- If the ball touches the back wall behind the paddle, the points are reset and the ball is reset.

(HINT: if the mouse y position is higher than the paddle y position, then the paddle should move upwards only in the y direction)
A.2.6 Pong 2

In this activity, you will take the 1 Player Pong activity that you completed in In-Lab 3 and continue working on it to make it a 2 player game. You will add the following functionality to your 1 player pong activity to make it 2 player:

- Add another paddle on the other side that will be controlled by pressing the directional buttons. The paddle should not go off the screen.

- Add a second score that will keep track of the second players points. The points will be added to the second player when the ball hits the wall behind the first players paddle.
EXAMPLE-BASED FEEDBACK RATING RUBRIC

The example-based feedback evaluation rubric is used in Chapter 6.

Note: The difference between the start code and end code shows whether the student should keep the code, or add a new feature. For example, if the start code has setVar, but end code does not, it suggests students remove setVar. For another example, if the start code does not have doAsk block, but the end code has, it suggests students add the doAsk block.

Relevance: Think of all the goals that the student may have had when they asked for the hint, based on what they have accomplished, the errors that they have and what they have left to do. How relevant is the suggested example pair to what the student is working on? Note: Consider the difference between the start code and end code.

1. The suggested example pair is irrelevant to the student’s code. For example, the example pairs suggest the student do things that the students do not plan to work on. The start code and end code is irrelevant to the student’s code. Or there is a huge gap between the start code and end code.

2. The suggested example pair is relevant to what the student will be working on but is less
relevant to their current work.

3. The suggested example pair is relevant to what the student is currently working on. For example, if a student is trying to draw a square and the suggested example pair draws the square. (The start code can be empty, and the end code can be drawing a square).

**Progress:** Will following the suggested example pair help students make progress towards a final correct solution? Note that if the student has a non-traditional solution, the hint should be judged with respect to the student's code, not an "ideal" solution. Also, consider the difference between the start code and end code. If the start code already reveals more correct information, such as showing a solution when the student only has a few code, then we still consider this could help the student make progress, unless the start code is incorrect or misleading.

1. The suggested example pair suggest an incorrect direction for the student or will not help the students make progress. For example, the end code may delete some useful code from the start code.

2. The suggested example pair is partly correct, but may help students move forward. For example, if the code is not in a custom block, but the final code should be in a custom block. For example, the students already have all the code suggested in the end code of the example pair and the student have correct value in the blocks.

3. The suggested example pair can help the student make progress towards a final correct solution. For example, the example pair shows a set of code that the student needs to make progress. For another example, the suggested example pair may have the same code as the student, but the values inside could help student progress towards a correct solution.

**Appropriateness & Interpretability:** As a tutor, would you present this example pair to the student? How likely is it that the example pair is generated by a tutor in response to the students' current code?

1. Unlikely, the suggested example pair is misleading or it needs further explanation for the novice.

2. Either unlikely or likely, the suggested example pair may be too vague for the student to understand it or contain too much information for the student to digest.

3. Likely, the suggested example pair is clear and easy to understand by a novice programmer who works on the assignment;

**Similarity:** How similar is the suggested example pair to the student's current code? Note: You are supposed to judge this category based on code similarity instead of code correctness. Please ignore the variable names and custom block names.
1. The start code of the suggested example pair looks different from the student’s code in important ways. The student may need extra time to relate the example code to their own. For example, the start code of the example pair has over 50% of code that is not present in the student’s code and the structure or the order of code is very different from the student’s code.

2. The start code of the suggested example pair shows a few (1-3) extra blocks compared to the student’s code. For example, the start code of the pair reveals one or two blocks that the student needs in the final solution but does not have for now.

3. The start code of the suggested example pair is similar to the student’s code. They have a similar code structure, order, and the start code of the example pair did not show multiple (more than two) unneeded blocks.