Investigating the Utility of Self-Explanation Through Translation Activities with a Code-Tracing Tutor

by

Maia Caughey

A thesis submitted to the thesis committee in partial fulfillment of the requirements for the degree of

Masters

in

Cognitive Science

Carleton University
Ottawa, Ontario

© 2022
Maia Caughey
Abstract

Programming is challenging for novices to learn. Code tracing is a foundational programming skill that involves simulating a program’s execution, tracking how variables change at each step. Translating lines of code into plain English explanations of what a given line of code does should allow students to better understand a program while code tracing. Translation can be characterized as a form of self-explanation, a general mechanism that helps learning by making inferences beyond the instructional materials. This thesis investigates whether self-explanation via translation of programming syntax to plain English helps students learn to code trace, in a between-subjects study ($N = 44$). We created two versions of a computer tutor that guides novice programmers through code-tracing problems. In the experimental condition, students were asked to translate lines of code during code-tracing activities, while in the control condition students were only asked to code trace. The experimental group performed significantly better on translation and code-generation questions, but the control group performed significantly better on code-tracing questions.
Acknowledgements

First and foremost, I would like to thank my supervisor Kasia for everything. Thank you for your endless support and patience throughout the entire process. Thank you for inspiring me from the very start, helping me develop and refine my research skills, and of course, appreciating the learning process. I am truly grateful.

Thank you to my committee, Dr. Alex Kelly, Dr. Robert Biddle, Dr. John Anderson. Thank you for your feedback and constructive input, and taking the time to read and understand my work.

Thank you to my friends Anderson Reid, Michael David II, Alex McGlasson, for providing me with friendship and support, and keeping me company during long work days. A special thank you to Corrine Staff for always being there support me throughout my personal growth, and constantly making sure that I spend some of my day laughing and smiling. Lastly, thank you Olivia McMurtry, without whom I would never have made it to grad school. Thank you for pushing me to be my best academically both academically and in my personal life, and always being there to talk it out. To all my friends, without your friendship, feedback, and advice I would not have been able to be where I am today.

Lastly to my family. Thank you to my parents for encouraging pursuing higher education of my interest, and your support of my decisions throughout my degrees. Thank you to my siblings, David and Ren. Thank you to you both for the social time, be it gaming or simply
even just talking about games, to give my brain a break. David, thank you for your wisdom about school and life, but also for your friendship and company. Sharing time through food, exercise, music, and shows was (and will continue to be) special to me. To Ren, your support in my life has been invaluable as I continue to grow as a person, thank you for being in my corner. Your caring nature and patience, being as present as I needed you has helped me in ways I cannot express.
# Contents

Abstract i

Acknowledgements ii

List of Tables vi

List of Figures vii

1 Introduction 1

2 Related Work 3

2.1 Code-Tracing Background 3

2.2 Supporting Learning of Programming with Tutoring Systems 8

2.3 Translating During Code-Tracing Activities 10

2.4 Self-Explanation 12

3 The Code-Tracing Tutor 20

3.1 Control Version of the Code-Tracing Tutor 20

3.2 Experimental Version of the Code-Tracing Tutor 23

3.3 Alternative Design Considerations: The Translation-First Design 27

3.4 Implementing the Two Versions of the Tutor 29
### Study Methods

4.1 Participants ......................................................... 31
4.2 Materials .......................................................... 32
4.3 Study Design and Procedure ....................................... 35

### Study Results

5.1 Introduction and Hypotheses ........................................ 37
5.2 General Descriptives ................................................ 38
5.3 Check for A Priori Differences ...................................... 39
5.4 Learning Results .................................................... 40

### Discussion

6.1 Learning Results by Question Type ................................. 48
6.2 Limitations and Future Work ........................................ 51

References .................................................................. 54

A Demographics Questions ............................................... 59

B Pretest ....................................................................... 60

C Posttest ...................................................................... 64

D Consent Form ................................................................ 68
List of Tables

5.1 Mean and standard deviation for pretest posttest, and gain (posttest − pretest) scores (by percentage) for each question type .................. 39

5.2 Mean and standard deviation for normalized gain scores (by percentage) for each question type .......................... 43
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A basic code trace (right) of a Python program (left)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>A basic code-trace activity for a Python program with translations</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>A completed code-trace activity on the control version of the tutor</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>An incomplete code trace table - a new box has unlocked after a correct answer for \textit{val} (green) is input into the code trace table</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>A completed code-trace activity with translations on the experimental condition version of the tutor (the right side of the interface is cropped for visibility reasons, see Figure 3.4 for the full interface)</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>A completed code-trace activity with translations on the experimental condition version of the tutor - purple box hides the translation solution until the code trace is complete</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Translation solution presented when students requested feedback by clicking on the reveal answers button for the problem in Figure 3.3</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>The first page of the translation-first design in which students only wrote the translation for all the lines of the program</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>The second page of the translation-first design in which students only completed the code-trace activity</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.8</td>
<td>A portion of a behaviour graph for one of the problems used in the experimental condition of the tutor</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>A slide from the video lesson that introduces students to the code-trace process with translations</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of a pretest problem that asked the student to write translations and code trace a short program</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Distribution of pretest scores (%) by question type and condition</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Distribution for normalized gain scores (%) by question type and condition</td>
<td>45</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Programming involves writing instructions called code that tell a computer how to solve a problem (e.g., sort a list of numbers). Computers cannot process human languages such as English, and thus code is written in specialized programming languages. Learning to program is a challenging process (Lahtinen, Ala-Mutka, & Järvinen, 2005; Fitzgerald, Simon, & Thomas, 2005; Cunningham, Blanchard, Ericson, & Guzdial, 2017; Jennings & Muldner, 2020) as it involves a series of novel skills (Xie et al., 2019).

Code tracing is a fundamental programming skill (Xie et al., 2019). Code tracing involves simulating the execution of a program and tracking how variables change at each step. Prior research has shown code tracing can improve learning of programming skills like code generation and code explanation (Hertz & Jump, 2013; Jennings & Muldner, 2020). Understanding the semantics of a program is required to code trace a program. If a student does not understand the program, they cannot simulate the execution of the program. A necessary step towards understanding involves knowing what a given program expression (or line) means. This can be accomplished by translating each program line and/or expression into a plain English explanation of what that code does. This translation grounds the code
in a familiar language that makes it easier to understand. Furthermore, while translating, students make inferences about the code that extend beyond the program syntax to better understand the individual expression or line of code. Self-explaining is effective for learning (Chi, Bassok, Lewis, Reimann, & Glaser, 1989; Renkl, 1999; Rittle-Johnson, Loehr, & Durkin, 2017) and involves generating inferences that extend beyond the information presented. Therefore, translation acts similarly to self-explanation. However, to date little research has tested whether translation benefits the learning of code tracing and code generation.

Tutoring systems have both been used to scaffold code-tracing and self-explanation processes (Conati & Vanlehn, 2000; Aleven & Koedinger, 2002; Nelson, Xie, & Ko, 2017; Jennings & Muldner, 2020). Scaffolding Tutoring systems provide guidance like feedback for problem solving activities. Tutoring systems allow for feedback to be provided to students in a streamlined and automatic fashion. Feedback has been shown to benefit learning (Anderson, Conrad, & Corbett, 1989; Margulieux & Catrambone, 2019). To our knowledge, no tutoring systems that include translation activities during code-tracing problems have been researched.

The present research took a step in addressing this gap. We created a computer tutor that guides novice programmers through code-tracing problems. The impact of self-explanation (translation) was evaluated in a between-subjects study with two conditions, one in which students were prompted to translate during a code-tracing task (experimental group), and one in which students completed a code-tracing task without being prompted to translate (control group). Learning was measured through a pretest and posttest with questions that targeted translation, code tracing, and code generation.

To our knowledge, no prior work has evaluated the utility of self-explanation in the form of translation activities during code tracing with a computer tutor. Before presenting the results of the study, we describe related work, details of the design of the tutoring system, as well as the study design.
Chapter 2

Related Work

2.1 Code-Tracing Background

A foundational skill students need to acquire when learning to program is code tracing (Xie et al., 2019). Code tracing entails simulating the execution of a program in a step-by-step fashion in order to predict variable values, the flow of execution, and the output produced (if any). A basic yet effective way to code trace involves “sketching” the code trace using paper and pencil (Cunningham et al., 2017). An example of this approach is illustrated in Figure 2.1. The short Python program (left) initializes two variables, \emph{counter} and \emph{total}, and updates their values in a loop until certain conditions are met. Students can use a code-trace table to track the values of the variables (Figure 2.1, right). Tracking variable values makes explicit how variables change as the program executes and helps to keep track of that process to correctly predict program output. To illustrate, Cunningham et al. (2017) analyzed students’ code traces produced as they completed a test that included a variety of programming problems (output prediction, code comprehension, ordering/fixing code, and code writing). Students were provided with scrap paper and asked to use it to sketch their
code traces while completing the test. The scrap paper was collected and analyzed. Students who generated explicit code traces, like the table in Figure 2.1 (right) performed better on the test than students who did not code trace. This data was collected from several sections of a university programming class. The way instructors and TAs code traced influenced how students code traced, suggesting students mirror what their instructors do, and so this is a skill that can be learned through instruction.

Other work also found benefits of code tracing. Lopez, Whalley, Robbins, and Lister (2008) analyzed student responses to an end-of-semester exam. Students answered a variety of programming questions, including programming basics, ordering lines of code to form a complete program, code tracing, code explanation, and code writing. There was a positive correlation between performance on code-tracing tasks and code-writing tasks. The analysis also revealed additional relationships between different programming skills that suggested a hierarchy of programming skills, where certain skills were precursors to others. Knowledge of programming concepts formed the bottom level of the hierarchy, and code tracing (among other programming skills such as code explanation) formed an intermediate level of this hierarchy, with code writing appearing in the higher levels. This suggests that code tracing is a more basic, fundamental programming skill. Venables, Tan, and Lister (2009) refined the
test questions used by Lopez et al. (2008) as well as the method of analysis. The results were consistent with Lopez et al. (2008), namely statistically significant and positive relationships between code-tracing, code-explanation, and code-writing skills. They also found indications that code-tracing competency preceded code-writing competency, but since their study was not experimental, more work is needed on this aspect. Several other researchers also reported a positive relationship between code tracing and code writing (Lister, Fidge, & Teague, 2009; Kumar, 2013). For instance, Kumar (2013) asked participants to predict the output of programs that contained if, if/else, or nested if/else statements by code tracing programs. Participants were given a pretest and posttest before and after the code-tracing session, respectively, that required them to write a program. On average, students had significant gains from pretest to posttest scores. This shows that practicing code tracing is beneficial for helping students learn to write code.

While it is an important and beneficial skill, code tracing is challenging for novice programmers (Lahtinen et al., 2005; Cunningham et al., 2017; Jennings & Muldner, 2020). Cunningham, Ke, Guzdial, and Ericson (2019) identified code-tracing difficulties through interviews. Participants were students in an introductory programming class. Twenty-six students were interviewed about the code traces they generated during a programming exam that assessed their knowledge of lists, loops, and conditional statements using code tracing and program generation questions. Students described reasons for why they did not code trace or why their code traces appeared different compared to the method they were shown by their instructors. Some students responded that they searched for patterns in the code that they recognized, and only code traced as a fallback if they had to, despite the fact that this approach is not a reliable way to predict program output. Students also described code tracing as unnecessary or excessive, making them go into more detail than they felt they needed to complete the problem, or consuming time that could be better spent on other
parts of the problem. Jennings and Muldner (2020) reported that scores on a pretest that assessed code-tracing skills were quite low (below 39%), even after an introductory lesson. After further practice working with a computer tutor, code-tracing performance improved but was still low (below 65%). In summary, code tracing is challenging.

Fitzgerald et al. (2005) reported on strategies that students used when posed with problems that required them to code trace. They analyzed data from twelve participants, evaluating what kinds of strategies they used when solving a variety of programming problems. Code walkthroughs (in which students traced through the code logic and/or updated variable values) were the most commonly used strategy. The authors indicated that problem structure appeared to affect the strategies used. While most students code traced, many of the code traces were incomplete or incorrect - similar issues were found by Cunningham et al. (2017). These findings suggest that novice students have a poor understanding of how to read and trace code.

2.1.1 Existing Support for Code Tracing

Since code tracing is a skill that novices struggle with, support is needed to help them learn to effectively code trace. Hertz and Jump (2013) observed that students in their introductory programming class struggled with building mental models needed for effective code tracing. In response, they re-designed their instructional approach, so it was centred on addressing this issue. Students were taught a lesson on a code-tracing topic, followed by a practice session where they worked in groups on code-tracing and code-generation problems. Students reported this approach as “helpful” or “very helpful”. Notably, this method had a significant positive effect on the lab grades. The lab materials focused on code writing, suggesting that instructor interventions on code-tracing topics increase performance on code-writing tasks. However, while observing their instructor trace code improved learning, students did not code
trace while completing lab work more than prior classes that did not receive code-tracing instruction.

Mayer (1975) conducted a series of three between-subjects experiments testing the utility of notional machine models to help students learn to code trace. A notional machine is an abstract representation of a computer, in this context used to illustrate what happens as a program executes. In the first experiment, participants were given code-tracing examples to study. There were four types of examples (each type was assigned to the corresponding condition): (1) model-based examples included a notional machine diagram with symbols to represent the computer actions during program execution at a high level; (2) text-based examples described the same computer actions without the diagrams, (3) flow-chart examples included the notional machine and flow chart; (4) text-based examples in conjunction with flow charts. Groups that studied examples with models (i.e., type 1 and 3) scored highest on code-tracing questions. Experiment 2 replicated the finding that model-based examples increased performance more than text-based questions. Experiment 3 evaluated the effect of prior knowledge in addition to practicing code tracing or code generation on test performance. Low prior knowledge students, who learned using either model-based examples paired with code-generation practice or text-based examples paired with code-tracing practice, performed well on tests. Students with high prior knowledge were impacted little by these manipulations in terms of their overall posttest scores.

The studies described above used traditional methods corresponding to classroom instruction and instructional materials to teach code tracing. Another approach involves program visualization tools. These tools show a step-by-step illustration of the high-level actions a computer takes as it executes a program, as well as the values of variables (and potentially other relevant program states). The use of these tools has been reported to be beneficial (Hoffswell, Satyanarayan, & Heer, 2018; Pérez-Schofield, García-Rivera, Ortin, & Lado, 2019).
However, visualization tools are traditionally designed for students with some programming experience (e.g., not novice programmers) and/or require more substantial learning and training to use effectively. Our focus is instead on mirroring how students code trace on paper since it requires minimal training for novice populations.

2.2 Supporting Learning of Programming with Tutoring Systems

Another approach for teaching code tracing and programming more generally involves tutoring systems. While tutoring systems have predominantly been developed for domains other than programming, such as math, physics, and foreign languages (Conati & Vanlehn, 2000; Aleven & Koedinger, 2002; Wylie, Koedinger, & Mitamura, 2010), there are notable exceptions corresponding to tutors for various programming activities (Anderson et al., 1989; Fabric, Mitrovic, & Neshatian, 2019; Hosseini et al., 2020). For instance, Anderson et al. (1989) designed and evaluated a computer tutor for the programming language LISP. This tutor aimed to help students learn to program using LISP by guiding them through writing code in LISP. Two versions of the tutor were compared, one that provided feedback on errors as students wrote LISP programs and a control tutor version that did not provide feedback. Students who worked with the tutor in the feedback condition made fewer repeat errors on future assessments.

We next describe recent work on the design and evaluation of tutoring systems for code tracing. Since this research is more relevant to the present work, more detailed descriptions are provided. Nelson et al. (2017) developed PLTutor, a computer tutor that guided students through code-tracing problems step-by-step. PLTutor was developed to help students learn programming semantics, and code-tracing skills. The tutor guided students through the
steps of code-tracing programs, highlighting three types of steps: (1) conceptual steps, for which the tutor walks the student through the program showing content directed at learning concepts, (2) execution steps, for which the student is shown content relevant to the execution of the program, and (3) assessment steps, for which students are prompted to fill in values throughout the program. The tutor allowed students to move forward and backward through program steps (with “Back” and “Next” buttons). The tutor scaffolded the different types of steps, for example, by adding a pause in which students were not permitted to move on from a conceptual learning step immediately, aiming to encourage reading. The tutor also included feedback, by providing a button that students could press that provided common misconceptions for inaccurate answers. Questions relevant to higher level code-explanation were included for some of the steps, however, these explanations were not explicitly produced by the user of the tutor, nor were they the core focus of the study. PLTutor’s was evaluated by comparing learning outcomes between two groups – one that interacted with PLTutor, and one that studied Codecademy materials. Students who worked with PLTutor learned more than the group that studied the Codecademy materials. The tutor was designed for computer science students. It is unclear whether these findings will generalize to students learning to code coming from different educational backgrounds as computer science students who may increased prior knowledge of the domain compared to from different backgrounds.

Jennings and Muldner (2020) implemented a code-tracing tutor that provided students with examples of worked-out code traces and prompted them to complete code-tracing problems. Two versions of the tutor were created, one that provided high scaffolding for code-tracing problems, and one that provided little scaffolding. The high-scaffolding version provided a code-tracing table, which guided students through the code-trace process, requiring them to enter the intermediate values of the variables in the code in the correct order, before inputting the final values of the variables. The table structure mirrored the code-trace
tables described in Cunningham et al. (2017) that students used while code tracing on paper. These features of the high scaffolding interface informed the design of the tutor used in the present study. Students were also provided with a plain English explanation of each program they had to code trace. Conversely, the low-scaffolding interface also provided a program to be code traced but only provided an open-ended scrap area for students to code trace. In this condition, students were only required to input the final values of the variables. A study was conducted to evaluate the tutor, with two factors: the amount of scaffolding, either high or low, and whether the tutor presented a worked-out example before completing a code trace or after completing a code trace (for a total of four conditions). All students completed a pretest, used a tutor to study four code-tracing examples and complete four code-tracing problems, and completed a posttest. Learning was highest when students received intermediate assistance from the tutor: they were either given the high-scaffolding interface and completed a code trace before seeing a worked-out example, or were presented with the low-scaffolding tutor and viewed a worked-out example before completing the code trace. Students provided with both forms of assistance (the high-scaffolding tutor and a worked-out example before completing a code trace) did not learn as much as those that were only provided one form of assistance.

Both these studies demonstrate that tutoring systems can help students learn to code trace, however, there is limited work to date and so more research is required.

2.3 Translating During Code-Tracing Activities

Understanding a program’s syntax and semantics is an important skill that is proposed to be an inherent part of code tracing (Xie et al., 2019). However, this is challenging because programs use a syntax and a language that is unfamiliar to novices. Xie et al. (2019) described
that line-by-line translations of program syntax into natural language (i.e., explanations) are an important first step for understanding the semantics of programs. A translation involves converting a statement written in a specialized programming language into an explanation of what the statement does in plain English, grounding the code in more familiar language to the student. These are low-level explanations that are different from explanations that describe the program as a whole (at a high level).

Some studies tested the utility of asking students to explain code at a higher level than we are targeting, because the explanations corresponded to describing what blocks of code do, or even overall program goals (Lopez et al., 2008; Whalley et al., 2006). To illustrate, Whalley et al. (2006) evaluated the reading comprehension skills of novice programmers. Participants were asked to explain what programs do at a high level (e.g., “What does this chunk of code do?” and “Explain the program”). These explanations targeted larger sections of programs, or entire programs. In contrast, during code tracing, students read line-by-line to trace variable values and program behavior. As noted above, Xie et al. (2019) suggested the step of translating a line into a plain English explanation during code tracing is a key skill for novice programmers.

To illustrate the translation and code-tracing process, Figure 2.2 shows a hypothetical example that simulates the process of code tracing a program. Before a student can code trace the program they need to understand it. Thus, beside each line of code is a green bubble with a hypothetical translation of the corresponding line (Figure 2.2, center). Once a line is translated, the code trace can continue, shown on the right for the program variables. To summarize, in this hypothetical example Python syntax has been translated to a plain English explanation, and only then is the code trace possible. Notably, translations only need to be performed the first time they are encountered in a program – once they are documented, students can refer to the translation (e.g., if the code occurs in a loop, only one translation is
needed).

```python
1 counter = 5
2 total = 1
3 while True:
4     counter = counter - 1
5     if total > 7:
6         break
7     total = total + counter
8     print(counter, total)
9 print("after", counter, total)
```

Figure 2.2: A basic code-trace activity for a Python program with translations

In theory, translating from a programming language to a human one during code tracing should be beneficial and potentially reduce shallow methods such as guessing and checking program output — guessing and checking involves rapidly entering solutions in tutoring systems until a correct one is produced (Baker, D’Mello, Rodrigo, & Graesser, 2010)). To date, however, no studies have tested if translation improves the skill of code tracing. Translation during code tracing is essentially a form of self-explanation, which is a general learning mechanism that we now describe.

2.4 Self-Explanation

Self-explanation involves generating inferences that extend beyond the information presented in instructional materials. Thus, translation is a form of self-explanation because in order to translate program statements, students need to make inferences about the program that
extend beyond what is written in the code. While to the best of our knowledge this form of self-explanation (i.e., translation in a code-tracing context) has not been studied, self-explanation has been extensively researched in other domains.

The seminal study by Chi et al. (1989) reported individual differences in terms of how students studied physics examples. In this study, eight students studied parts of a physics textbook, including three worked-out examples of physics problems (dealing with Newton’s laws of motion). Some students, labelled the “good” explainers, self-explained when studying the examples. For instance, they rederived example solution lines using the principle that generated the line. This principle was not shown in the example solution, thus requiring students to test their knowledge of the target domain. Other students, labelled “poor” self-explainers, paraphrased the examples, producing few explanations. The “good” self-explainers learned more than the “poor” self-explainers. Renkl (1999) replicated these results in the domain of mathematics, also identifying differences between students’ self-explanations. For instance, “good” students, namely ones who learned more, anticipated the next steps in their explanations and/or described relations between example operators and overall goals, whereas poor students rarely self-explained. Thus, in summary, self-explanation is good for learning, but some students fail to spontaneously self-explain.

More recently, Rittle-Johnson et al. (2017) confirmed that self-explanation increases learning through a meta-review in the mathematics domain. They propose that self-explanation is beneficial because: (a) it allows students to integrate new knowledge with prior knowledge; (b) when there is conflict between new information and existing knowledge, self-explanation allows students to resolve these incongruencies; (c) self-explanation guides students’ attention to structural features, ones relevant to the solution of to be learned content, making their knowledge more generalizable. Rittle-Johnson (2017) proposed four guidelines for promoting self-explanation (informed through the meta-analysis conducted):
• Guideline 1: Scaffold explanations via training on self-explanation or structuring self-explanation responses by providing student alternatives to free-response self-explanations (such as filling in blanks in prewritten self-explanations or selecting explanations from menus or glossaries).

• Guideline 2: Design explanation prompts so they do not sacrifice attention to other important content.

• Guideline 3: Prompt learners to explain correct information.

• Guideline 4: If erroneous examples/text is included, prompt learners to explain why incorrect information is incorrect if there are common errors or misconceptions.

These guidelines were developed from the findings of the meta-review and reflect, as Chi et al. (1989) and Renkl (1999) demonstrated, that self-explanation is not something all students do spontaneously.

2.4.1 Support for Self-Explanation

Since self-explanation is beneficial for learning but not all students are spontaneous self-explainers, support is needed to encourage self-explanation. Common forms of support include prompting (explicitly asking students to self-explain), feedback (telling students if and how correct their explanations are) and training (teaching students about self-explanation). We focus primarily on prompting since that is most relevant to the present work.

Prompting is a common method for eliciting self-explanation (Aleven & Koedinger, 2002; Rittle-Johnson et al., 2017; Margulieux & Catrambone, 2019). Chi, De Leeuw, Chiu, and LaVancher (1994) tested the utility of prompting for self-explanations. Two groups read a passage about the human circulatory system. In the experimental condition, students
were prompted to self-explain after having read each line of the passage, explaining what they understood about the sentence. The control group was not asked to self-explain. The experimental group learned more than the control group. Furthermore, within the prompted self-explanation group, students who generated many self-explanations learned more than students who only generated a few self-explanations. Recent work has confirmed that providing scaffolding for self-explanations benefits learning (Renkl, 1999; Rittle-Johnson et al., 2017). Scaffolding can take the form of prompts, like “Explain how examples 1 and 2 are similar” (De Koning, Tabbers, Rikers, & Paas, 2011).

While the majority of research on prompting for self-explanation has been carried out in domains other than programming, there are notable exceptions. Margulieux and Catrambone (2019) conducted a study involving novices using Android App Inventor. In this study, students used a “drag and drop” programming language to learn programming concepts through worked examples that included self-explanation prompts. The prompts asked students to explain the subgoals for the example solution. Subgoals are sub-steps of procedures used to solve problems. For example, two lines of code setting the width of a canvas and the height of a canvas fulfill the subgoal “set canvas properties”. To test the utility of asking students to self-explain prompts, in some conditions, students were given example programs that already had the sub-goals while in others, they had to infer the subgoal via self-explanation. The study also manipulated the presence of hints and feedback for correctness. The key finding was that students who had to generate subgoals and who either received feedback or received hints (but not both) learned significantly more than students in the passive conditions where the subgoals were provided and did not have to be self-explained. Thus, encouraging self-explanation led to more learning and some assistance for self-explanation was beneficial (but not too much assistance).

Traditionally, work was focused on encouraging self-explanation through traditional
materials such as pen and paper. Recent work has moved to a digital format but maintained a similar setup to traditional methods, imitating how students might self-explain on pen and paper, or by having experimenters prompt participants to self-explain. Tutoring systems are another method of scaffolding self-explanation. As was the case for the prompting work described above, the majority of tutoring systems target domains other than programming. We begin with a brief description of several examples of this work.

Conati and Vanlehn (2000) designed and evaluated a tutoring system that provided worked-out examples of physics problems. To encourage students to self-explain the example solutions, the tutoring system provided general prompts reminding students to self-explain as well as a tool to derive the principles (i.e., physics rules) that produced the example solution lines. Students were provided with correctness feedback on their self-explanations. The system was evaluated using an experiment with university students and two conditions. In the experimental condition students were provided with the self-explanation scaffolding and feedback, while in the control condition, students studied the example without access to the tool or feedback. Students in the experimental condition learned more than in the control condition. Aleven and Koedinger (2002) also demonstrated the utility of self-explanations in the context of a tutoring system that provided geometry problems. In two classroom experiments, they found that students who self-explained their steps as they solved problems with the computer tutor learned more than students who did not explain. As a third example, Wylie et al. (2010) evaluated a tutoring system for grammar learning. In an experimental study, they evaluated two instructional contexts, one targeting declarative knowledge in which students self-explained in a structured way by selecting the correct article to be used in a sentence (i.e., choosing whether to use a, an, the, or no article), and the other, targeting procedural knowledge in which students were asked to free-response self-explain why the article used in the sentence was correct. In both conditions students learned, demonstrating
the utility of two types of self-explanations.

More recently, research on promoting self-explanation through tutoring systems has included the domain of programming. Most work has targeted code generation. For instance, Fabic et al. (2019) developed a mobile tutor app that helped students to learn Python. The tutor presented incomplete programs that students were asked to complete. After completing a given line of code, students were prompted to self-explain that line. To help them produce the explanation, they were provided with several possible explanations that elaborated on the given line (e.g., “Which of the statements is CORRECT about the incomplete Line Of Code you just answered?”) – students had to select the correct explanation (Fabic et al., 2019). This full version of the tutor was compared to one that did not prompt for self-explanation in an experimental study. The prompted group had significantly higher learning compared to the unprompted group (as measured by normalized gain). As a second example, Hosseini et al. (2020) designed and evaluated a computer tutor that provides program construction examples with prompts to encourage constructive reasoning through “challenge me” buttons. These “challenge me” activities require the students to engage in program construction exercises, dragging and dropping blocks of code as well as reorganizing lines of code. This version of the tutor was compared to a control version that presented textbook-style example activities without “challenge me” prompts. They found that work with the experimental group that worked with the “challenge me” activities learned more than the control group.

Tamang, Alshaikh, Khayi, Oli, and Rus (2021) designed a computer tutor that provided one of two forms of prompts: (1) free self-explanation prompts (an open text box with space to describe what the program goals and outputs); (2) the Socratic method (in which students were given prompts about the program such as “what are the values of i during the execution” and “how many times will the loop be executed” by a tutor). The domain was Java code-tracing problems. A third group was included as a control group that was asked only to
predict the outcome of the programs. Both experimental groups learned significantly more than the control group. Furthermore, between the two experimental groups, the group that learned using the Socratic method learned significantly more than the prompted condition.

Not all studies found that prompting for self-explanation benefited learning. Kumar (2014) developed a tutoring system that presented code-tracing problems. The tutoring system had two versions, one for the experimental condition in which students were prompted to self-explain, and one for the control conditions in which they were not. The tutor presented code-tracing problems and if a student submitted an incorrect solution, the system presented a worked-out example showing the correct solution. In the experimental condition, self-explanation prompts that were integrated into the example. The prompts asked students to self-explain as they reviewed the worked examples by selecting correct answers to fill in a blank step-by-step solution. There was no significant difference in learning between the control and experimental groups. The author speculated that factors other than self-explanation may have hindered learning. Students used the tutor in their own time and were only provided with completion credit. Thus, perhaps the self-explanation task was completed as quickly as possible. Therefore, it may be necessary for students to have additional support while learning from self-explanations with a computer tutor.

2.4.2 The Present Study

In sum, learning to program is challenging. Both tutoring systems and self-explanation have been shown to be helpful for helping students to learn. However, there is no research that has investigated the utility of self-explanations for code-tracing skills. The present work investigates if prompting for self-explanation by having students translate programming language code into plain English during code-tracing activities improves learning of programming skills. The activities are embedded into a tutoring system we built using the CTAT framework
(Aleven et al., 2016). We next describe this tutor.
Chapter 3

The Code-Tracing Tutor

3.1 Control Version of the Code-Tracing Tutor

We created a tutoring system for code tracing and used it to test the impact of translation activities. This tutoring system contained programs written in the Python programming language. Two versions of the tutoring system were created - we begin by describing the standard code-tracing tutor (also referred to as the control version since it was used by the control group in the study we conducted). This version of the tutor was used by the control group – it included four code-tracing problems without the translation component. The tutor presented one code-tracing problem per screen (see Figure 3.1). Each screen included a brief Python program (Figure 3.1, left) and instructions to code trace the program by simulating its execution (Figure 3.1, top). Students were asked to keep track of the values of the program’s variables using in the code-trace table (Figure 3.1, right).

To illustrate, Figure 3.1 shows a completed code trace. Before the loop, the values of \textit{val} and \textit{res} are 5 and 10, respectively (see the first row of code-trace table in Figure 3.1). These values are subsequently updated to 4 (line 7) and 6 (line 8) as a result of the first
while loop iteration (see the second row of code-trace table in Figure 3.1). Upon simulating the execution of the second loop iteration, the values of val and res are updated to 3 and 3, respectively (see the last row of the code-trace table in Figure 3.1). In the last loop iteration, the condition on line 5 in the program is satisfied and thus the program breaks out of the loop at line 6. The tutor does not ask students to show program output as the focus is on tracing variable values. To move on to the next problem, the student presses the “Done” button (Figure 3.1, bottom-right).

The tutor provides scaffolding for code tracing in three ways. First, the code trace table specifies which variables need to be traced. The table design is based on a format used by effective code tracers (Cunningham et al., 2017). The second form of scaffolding corresponds
Figure 3.2: An incomplete code trace table - a new box has unlocked after a correct answer for val (green) is input into the code trace table

to providing students with feedback. Prior work shows that feedback improves learning (Anderson et al., 1989). Feedback can be provided in many ways – for the present work, the tutor provides the student with immediate correctness feedback on code-tracing by colouring their entries green or red for correct and incorrect entries, respectively.

The third form of scaffolding corresponds to guidance through the code trace, by only allowing the student to input an answer for a step after the previous step has been correctly completed. Initially, all input boxes in the table except for the first box are “locked” (they are visually greyed out and cannot be clicked on or edited). Once a student correctly inputs an answer, the next box unlocks (see Figure 3.2). This method of code tracing was used in the introductory lesson about code tracing provided to students before they had to use the tutor, and so should be familiar to students. This design aimed to guide students during answer generation as well as required them to show their work for each step. Prior research has shown that many students had incomplete code traces, which was associated with less
success on tests (Cunningham et al., 2017). This design aims to mitigate this by eliminating incomplete traces.

### 3.2 Experimental Version of the Code-Tracing Tutor

We now describe the experimental version of the tutor. Like the control version described above, the experimental version presented the student with a program, and a code-tracing table was used to enter a step-by-step code trace. However, the experimental version additionally asked students to provide a self-explanation of each line of Python code by translating the code into a plain English explanation of what that code does. The tutor provided guidance for the translation process: it required students to generate the translation before making updates to the code-trace table (if applicable – since this program involves a loop, translations for subsequent loop iterations beyond the first are redundant and therefore not required). The rationale for this design choice is explained below.

To illustrate, Figure 3.3 shows the tutor interface for the experimental condition with a completed code trace. Initially, all the code-trace table boxes and translation boxes are blank and locked (except for the first translation box next to the first program line, which is blank and not locked). To begin code-tracing, a student needs to first enter the translation for the first line of the program. Since only variable updates are included in the code-trace table, the next box that unlocks is the second translation box (see orange bubble labelled “1” in Figure 3.3), allowing the student to enter the translation for that line of code (e.g., “assign the value 5 to the variable val” in Figure 3.3). Once the translation is entered, because there is a variable update, the corresponding box in the code-trace table is unlocked and the student can enter the value of the variable val (see orange bubble labelled “2”, Figure 3.3). The student can then move to the next line in the program, translate that line (see
Figure 3.3: A completed code-trace activity with translations on the experimental condition version of the tutor (the right side of the interface is cropped for visibility reasons, see Figure 3.4 for the full interface)

the orange bubble labelled “3” in Figure 3.3), and enter the next value into the code-tracing table (see the orange bubble labelled “4” in Figure 3.3), and so on.

This version of the tutor provided the same forms of scaffolding for code tracing as the control interface (i.e, code-trace table, feedback for correctness on code-tracing entries, guidance by requiring a certain order of entries, as described above). As far as the translations (the novel aspect of this interface), the translations were scaffolded by including separate boxes for each one. Each box corresponds to the translation of a single line of code since they
Figure 3.4: A completed code-trace activity with translations on the experimental condition version of the tutor - purple box hides the translation solution until the code trace is complete were required for each line. This design aimed to remind students that code tracing involves a translation at each line of the program. Recall that the tutor required students to generate the translation before they could update the related code-trace table entry. This was intended to encourage understanding of the program at a semantic level. For example, the expression \( val = val - 1 \) can be translated to something like “reduce the value of variable val by 1” (see the orange bubble labelled “5”, Figure 3.3) rather than the more shallow explanation “val is changed from 5 to 4” as the latter only focuses on the current value of a variable. In general, the hope was that translations would reduce errors during code tracing because students will know what the program means.

Due to challenges with natural language parsing, immediate feedback on the translations was not provided, and any input submitted for a translation unlocked the next box. Several methods of providing feedback were considered, including using natural language processing
<table>
<thead>
<tr>
<th>CODE</th>
<th>TRANSLATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print(&quot;Number 1&quot;)</code></td>
<td>Prints the string “Number 1” to the screen</td>
</tr>
<tr>
<td><code>val = 5</code></td>
<td>Assign the value 5 to the variable val</td>
</tr>
<tr>
<td><code>res = 10</code></td>
<td>Assign the value 10 to the variable res</td>
</tr>
<tr>
<td><code>while True:</code></td>
<td>Complete lines of code inside the loop</td>
</tr>
<tr>
<td><code>if res &lt; 5:</code></td>
<td>Check if the variable res is less than 5</td>
</tr>
<tr>
<td><code>break</code></td>
<td>Exit the loop</td>
</tr>
<tr>
<td><code>val = val - 1</code></td>
<td>Lower the value of the variable val by 1</td>
</tr>
<tr>
<td><code>res = res - val</code></td>
<td>Lower the value of the variable res by the value of the variable val</td>
</tr>
<tr>
<td><code>print(val, res)</code></td>
<td>Print the value of the variables val and res to the screen</td>
</tr>
</tbody>
</table>

Figure 3.5: Translation solution presented when students requested feedback by clicking on the reveal answers button for the problem in Figure 3.3

... tools to match text, keywords matching, and other basic methods of analyzing text responses. However, in the end, the tutor’s ability to provide immediate, detailed translation feedback turned out to be beyond the scope of this project. To provide translation feedback, the tutor gave students the option to view the correct translations for the program by clicking on the large, coloured box to the right of the interface (Figure 3.3, right). Once clicked, the “model” translation solution was shown (see Figure 3.5).

Translation feedback was delayed, in that the translation solution could only be viewed once the entire code-trace problem was completed. Students could edit and correct their translations answers before moving on to the next question.
3.3 Alternative Design Considerations: The Translation-First Design

Alternative designs for the tutor were considered during development, which we will now describe briefly. Whether the tutor should present translation activities before code tracing instead of concurrently while code tracing (as is currently done) was considered. For this “translation-first” approach, the student would generate translations for each line of code for the full program. Once completed, only then would they complete the code-trace activity.

Mock-ups of what this design might look like were developed. Figure 3.6 shows a screen with entry boxes for translating each line of the program. Note that code tracing is not part of this interface. Once the translation was complete (all translation entries generated), the student would click the arrow in the bottom right corner to move on to the code-trace activity on a separate screen.

Figure 3.7 shows a mock up of the translation screen (essentially the control interface). Note that the translations are not included in this screen. The student could return to their translations by clicking the arrow to the bottom right of the page. Each page of this design also included a “see answer” button in order to provide feedback to the student. A limitation of the translation-first approach is that this format imposes a linear reading of the code, i.e., reading the program from top to bottom. This does not necessarily match the execution of the code, such as when loops are included in the program. When loops are included in the code, it is common that the flow of execution returns to program locations that may have already been translated, which is an aspect of code tracing that would be ignored in this format. Furthermore, this format may increase cognitive load as students may need to switch between the pages to remember what a certain line of a program meant. Increased cognitive load can hinder a student’s ability to learn (Sweller, 2011). Due to these two limitations, the
Figure 3.6: The first page of the translation-first design in which students only wrote the translation for all the lines of the program

Figure 3.7: The second page of the translation-first design in which students only completed the code-trace activity
translation-first design of the tutor was not selected.

3.4 Implementing the Two Versions of the Tutor

Recall that the control tutor included code-tracing activities without translation, while the experimental tutor required translation prior to a code trace. The two versions of the tutor (control and experimental) were developed using the CTAT authoring tools (Aleven et al., 2016). The interfaces were created using the CTAT HTML Editor, which creates HTML and CSS files for the tutor and is used to design the core interfaces for the tutors. The developer of the tutor can select features of a webpage to include in the tutor and customize the features to design an interface that users will see and work with. The tutor behaviour (e.g., feedback correctness, interface behaviour related to locking/unlocking of components) was specified using the CTAT authoring tool. This tool allows a human author to specify tutor actions, by recording actions that the human author makes on the tutor interface. The recorded series of steps is converted into a flow chart called a behaviour graph. When uploaded to a platform that integrates the behaviour graph with the corresponding interface developed in CTAT HTML Editor, the tutor can be disseminated in a browser and used by students.

Using CTAT for HTML and FLASH, the behaviour graphs can be edited to include a variety of customizations that enhance the quality of the tutor. We used the behaviour graph to control the order in which boxes on the interface are interacted with and filled in. CTAT for HTML and FLASH allows the author to switch boxes to be editable when a certain condition has been met. This is the basis of how the locking and unlocking of boxes was implemented. Behaviour graphs also allow the author to indicate whether an input is correct at specific steps, which is how feedback was provided for code-trace table entries. To illustrate, Figure 3.8 shows a snippet of a behaviour graph. The second box (with green text)
indicates that an input box labelled \textit{transl1} will receive input (this is the first translation box of the interface). The asterisk (*) indicates that the input can be any input (i.e. the student can write anything here, and no feedback will be provided). After that input, the tutor will be in a new state named \textit{state1}. The next box after the \textit{state1} box indicates that an input box labelled \textit{transl2} changes from being un-editable to being editable (this is another translation box and is the next step of the code-trace activity). This brings the tutor again to a new state. Every time the interface changes, due to an input from the user, or a new box changing from un-editable to editable, the behaviour graph has a corresponding action box (in green text) and state box (black text). The behaviour graph continues this pattern until the last two boxes, the see sample example answers box and the “Done” button. Students must click on the large coloured box before the “Done” button is unlocked, and only then can the student click on that box to finish the question.

![Figure 3.8: A portion of a behaviour graph for one of the problems used in the experimental condition of the tutor](image)

The two tutors were deployed on the platform TutorShop, which made the tutors available in a Web browser.
Chapter 4

Study Methods

We now describe the study we conducted to evaluate the code-tracing tutor. This project was reviewed and cleared by the Carleton University Research Ethics Board B.

4.1 Participants

The study participants were 44 individuals (37 female, 6 male, and 1 demi-femme) recruited using class announcements in a first-year university programming class (CGSC 1005), the SONA online recruitment system available to students in a first-year university class that provided a broad overview of cognitive science (CGSC 1001), as well as social media advertising via Facebook (two groups, one for cognitive science students, and one for research participation opportunities), and word of mouth. Participants who signed up through SONA or the CGSC 1005 class announcement received a 2% bonus course credit for completing the study. Individuals who did not participate for course credit received a $25 compensation instead. To be eligible, participants either must have had no prior programming experience or had taken at most one university-level programming course. Participants who had taken more than one university programming course were not eligible for the study, to avoid ceiling effects on the
learning measure.

4.2 Materials

A brief programming lesson was developed to provide students with an introduction to fundamental programming concepts. The lesson corresponded to a 20-minute video showing a narrated slideshow. The topics covered were variable assignment, integer and string data types, basic conditional statements, and while loops. While there are two common types of loops in Python including while loops and for loops, the lesson focused only on while loops because they are more basic (based on teaching experiences at Carleton University that suggested that students find for-loops more challenging). The while loops were of the ‘while True’ variety and conditional statements inside the loop determined when the looped stopped (i.e., when a particular condition was met). The lesson included examples of brief programs in the Python language involving these constructs. The focus was on the instruction of code-tracing skills, with an emphasis on translating Python code to plain English descriptions.

Figure 4.1 below shows a slide used in the lesson to illustrate how to code trace a simple program. The program is on the left, with corresponding translations of each program line in the center (see green speech bubbles, Figure 4.1), and a code-trace table on the right. The instructor in the lesson walked students through code tracing the program, revealing the steps one at a time using animations built into the slide. First, a translation bubble was revealed by the instructor, and the variable value was revealed in code-trace table only after the translation appeared for line the corresponding to the variable update.
Figure 4.1: A slide from the video lesson that introduces students to the code-trace process with translations

A brief demographics questionnaire was used to obtain information on the participant’s major, gender, and experience with programming courses, see Appendix A.

A pretest and posttest were used to measure domain knowledge and learning. The tests were isomorphic in that they had the same number of questions and the same question content, just with different variable names (see Appendix X for the pretest). Each test included seven questions.

The first three questions showed a short Python program consisting of 3-5 lines and asked for an explanation of each program line as well as what the program prints to the screen. The questions dealt with assignment, if-conditional statements, and user input. The remaining questions were similar but involved longer programs (7-10 lines of code). The final question was a code-generation question included as a transfer question to check if students would transfer code-tracing knowledge from the intervention to code generation. All code-trace questions had only one correct answer, while the translations and the code-generation question had multiple correct answers. A grading rubric was developed for the pretest and posttest, the maximum number of points on each was 44.5.
Example Pretest Question

<table>
<thead>
<tr>
<th>Q1</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen.</td>
</tr>
</tbody>
</table>

Program:

1. `outcome = 1`
2. `outcome = outcome + 10`
3. `print(outcome)`

**Line 1 explanation:**

Your answer

**Line 2 explanation:**

Your answer

**Line 3 explanation:**

Your answer

**What does the program from above (copied below) print to the screen?**

1. `outcome = 1`
2. `outcome = outcome + 10`
3. `print(outcome)`

Your answer

Figure 4.2: An example of a pretest problem that asked the student to write translations and code trace a short program
4.3 Study Design and Procedure

We used a between-subjects design with two conditions – an experimental condition and a control condition. In the experimental condition students completed translations while code-tracing programs. In the control condition, students code-traced programs without translating. Participants were assigned to a condition in a round-robin fashion. The procedure for the two conditions was identical, the only difference being which version of the tutor the participant interacted with.

The study sessions were conducted individually (one participant per session) through Zoom, which was dictated by the ongoing COVID-19 pandemic. When participants signed up for the study, they received an email confirming their participation as well as requirements for the study (such as internet access, and a computer). A reminder email with the Zoom meeting link was sent the day before or the morning of the study. Each participant was assigned a participant ID, used instead of their name for all study data. The study took no more than two hours to complete. For the first part of the Zoom meeting, the researcher’s camera was on. Participants were not requested to turn their cameras on, but some participants voluntarily had their cameras on at the beginning of the meeting or turned them on.

First, the researcher welcomed the participant and provided them with a link to a consent form. Once signed, participants were given an overview of the study, outlining that they would watch a video lesson, complete a programming questionnaire, complete practice problems, and then complete another questionnaire. Once ready to begin, participants were provided with the link to the video lesson. They were asked to watch the lesson without pausing or rewinding the video. Participants were also told that they could take notes, but it was not required. During the video lesson, the researcher’s camera was turned off to reduce distractions. After finishing the lesson participants were provided with a link to the demographic questionnaire and the pretest. The demographic questionnaire was placed at the beginning of the pretest.
Participants were informed they had 20 minutes to complete the pretest questionnaire (with a five-minute grace period). Participants were asked to notify the researcher when they finished the questionnaire if they finished in less than the allotted time. Next, participants were given a five-minute break and then provided a link to the TutorShop website where the tutor was stored, as well as a username and password. Once they logged in, they were given a brief five-minute introduction to the tutor by the researcher, including a demonstration on how to use the tutor. This demo used a problem seen in the video lesson the participants watched at the beginning of the study. The researcher completed the code-tracing activity, demonstrating the full use of the tutor from the beginning to the end of a problem. During this stage, participants could ask questions about how to use the tutor. Once the tutor demonstration ended, participants used the tutor to solve four problems. Participants were given 40 minutes to complete the problems (with a 10-minute grace period). Participants were instructed to complete the problems at their own pace. Participants were asked to notify the researcher if they finished the problems in less than the allotted time. Following the practice phase, participants were provided with a link to the posttest and given 20 minutes to finish (with a five-minute grace period). As the final step, the researcher debriefed the participant, answered any questions about the study, and for participants other than SONA, obtained information needed for compensation (i.e., email address – SONA participants were already in the SONA system).
Chapter 5

Study Results

5.1 Introduction and Hypotheses

The following research question guided the analysis:

1. Does engaging in explicit translation activities during code tracing improve learning over standard code tracing without translation?

To answer the research question, a measure of learning was needed. As the first step, the pretest and posttest were graded by the thesis author using a grading scheme. There were three types of questions on the test, namely translation questions, code-tracing questions, and a code-generation question (this was the transfer question, as the lesson did not cover program generation). Both tests were graded out of 44.5 points: 19 points total for translation questions (40% of the total), 17.5 points total for code-tracing questions (39% of the total), and 6 points for code-generation questions (13% of the total). For the results, we report average percentage score by question type.

We hypothesized that the experimental group would perform better than the control group on translation questions, given that it had practiced translation while working with
the computer tutor and received feedback from the tutor on translations. We hypothesized that the experimental group would also perform better on code-tracing questions given that translation is theorized to be an integral part of code tracing (Xie et al., 2019). We did not have a hypothesis related to code-generation performance.

For the analysis, we used both descriptive and inferential statistics. Null Hypothesis Significance Testing was used for inferential stats, the primary tests being independent samples t-tests. Violations of homogeneity of variance were tested for, using Levene’s test, and if violations were detected we report the adjusted statistics.

5.2 General Descriptives

We begin with the descriptive statistics, shown in Table 5.1. The pretest performance depended on the question type, with the highest scores for translation questions, followed by code-tracing questions, with the lowest scores on code-generation questions. Despite the condition assignment strategy used, by chance, descriptively the experimental group had higher pretest scores for all three question types as compared to the control group. The pretest to posttest gains depended on the condition. Notably, the control condition for translation questions had slightly negative gain scores, meaning that their performance got worse after the intervention. In contrast, the experimental group’s translation scores increased from pretest to posttest. However, the experimental group’s code-tracing scores essentially remained the same between pretest and posttest, while the control group’s scores on these questions improved.
<table>
<thead>
<tr>
<th>Question Type</th>
<th>Control $n = 22$</th>
<th>Experimental $n = 22$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M$ (SD)</td>
<td>$M$ (SD)</td>
</tr>
<tr>
<td>Translation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pretest (%)</td>
<td>66.99 (23.60)</td>
<td>74.88 (19.49)</td>
</tr>
<tr>
<td>Posttest (%)</td>
<td>63.04 (28.71)</td>
<td>78.71 (22.58)</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>-3.95 (15.70)</td>
<td>3.83 (8.11)</td>
</tr>
<tr>
<td>Code-Trace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pretest (%)</td>
<td>50.78 (23.43)</td>
<td>68.83 (27.99)</td>
</tr>
<tr>
<td>Posttest (%)</td>
<td>66.75 (31.24)</td>
<td>68.70 (27.82)</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>15.97 (20.93)</td>
<td>-0.13 (8.93)</td>
</tr>
<tr>
<td>Code-Generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pretest (%)</td>
<td>23.49 (28.48)</td>
<td>38.64 (38.28)</td>
</tr>
<tr>
<td>Posttest (%)</td>
<td>31.44 (34.69)</td>
<td>57.20 (39.37)</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>7.96 (21.59)</td>
<td>18.56 (27.93)</td>
</tr>
</tbody>
</table>

Table 5.1: Mean and standard deviation for pretest posttest, and gain (posttest – pretest) scores (by percentage) for each question type

5.3 Check for A Priori Differences

Our next step involved checking if the differences in pretest scores between the two conditions were reliable, as that would indicate a priori differences between conditions. In preparation for this analysis we checked the distribution of the pretest scores for each question type, shown in Figure 5.1. There were three outliers in the experimental condition for the translation questions (no outliers for the other question types).

Aside from outliers, there are a couple of notable features regarding the distributions. In general, the range of scores for all three question types and both conditions is very large. For the translation scores, the control group has greater variability than the experimental group,
with lower scores as indicated by the width of the lower whisker. For the the code-generation questions there is slightly higher variability in the experimental group compared to the control group. Also, note the odd distributions of the code-generation scores. The lack of a lower whisker and the interquartile box sizes indicates there is less variability in the lower 50% of scores in both the experimental and control groups compared to the upper 50% of scores.

To check for a priori differences in pretest scores between the two conditions, an independent samples t-test was used. The experimental group performed significantly better on code-trace questions, $t(42) = 2.32, p = .025, d = 0.70$. The pretest differences between conditions for the other two question types were not significant (translation pretest scores: $t(42) = 1.21, p = .233, d = 0.37$; code-generation pretest scores $t(42) = 1.49, p = .144, d = 0.50$).

Since outliers were present for the translation pretest questions (see Figure 5.1, left), they were removed, and the analysis was re-run. The pattern remained the same, as the difference between conditions on translation questions remained not significant. Unless otherwise noted, the outliers were kept in the analysis.

### 5.4 Learning Results

To check if participants learned overall (regardless of condition or question type), a paired samples t-test was run after checking for outliers in the pretest to posttest difference scores (there were none). Overall participants improved from pretest to posttest ($M_{gain} = 19.81, SD = 32.75$), $t(43) = 3.57, p < .001, d = 0.54$.

To check for conditional differences, normalized gain was used as the measure of learning
Figure 5.1: Distribution of pretest scores (%) by question type and condition
(Coletta & Steinert, 2020). The formula to calculate normalized gain is as follows:

\[
\frac{\text{Posttest}(\%)-\text{pretest}(\%)}{100\% - \text{pretest}(\%)}
\]

Normalized gain characterizes how much a student learned relative to how much they could have learned. This is accomplished by an adjustment of the gain score with pretest scores – enabling a more fair comparison between groups. To illustrate, suppose two students both gained 10 points from pretest to posttest scores but one student had a low pretest score, and the other student had a high pretest score. Despite having the same raw gain, the student who had a lower pretest score gained less relative to how much they could have, whereas the student with the higher pretest score gained a large amount relative to what they were able to gain. Accordingly, the student with the lower pretest score will have a lower normalized gain score, and the student with the higher pretest score will a higher normalized gain score. While using normalized gain does not eliminate a priori differences, it aims to account for them through this adjustment of pretest scores. Note that if a participant has a perfect score on their pretest (i.e., a ceiling score), the normalized gain is undefined. These instances were excluded on a per analysis basis. \(^1\)

The descriptives for the normalized gain scores for each question type and condition are shown in Table 5.2, with and without outliers. We begin by describing the results considering all data including outliers (Table 5.2, left).

For the translation questions, the control condition performed worse on the posttest than on the pretest as indicated by a negative normalized gain (note that the raw gain scores shown in Figure 5.1 indicated a smaller negative gain, but the normalized gain penalizes losses

\(^1\)If there was a ceiling score for a give question type, only data for that question type was excluded, rather than excluding all of the participant’s scores from analysis. Given that they are at the ceiling on the pretest, which produced slight variations in the degrees of freedom, the normalized gain formula also results in a undefined error due to dividing by zero.
<table>
<thead>
<tr>
<th>Question Type</th>
<th>Control</th>
<th>Experimental (outliers removed)</th>
<th>Control</th>
<th>Experimental (outliers removed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M \ (SD)$</td>
<td>$M \ (SD)$</td>
<td>$M \ (SD)$</td>
<td>$M \ (SD)$</td>
</tr>
<tr>
<td></td>
<td>$n = 22$</td>
<td>$n = 22$</td>
<td>$n = 21$</td>
<td>$n = 22$</td>
</tr>
<tr>
<td>Normalized Translation</td>
<td>-23.30 (77.55)</td>
<td>20.30 (39.47)</td>
<td>-13.70 (64.68)</td>
<td>20.30 (39.47)</td>
</tr>
<tr>
<td>Gain (%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normalized Code-Trace</td>
<td>$n = 22$</td>
<td>$n = 19$</td>
<td>$n = 22$</td>
<td>$n = 15$</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>38.16 (46.07)</td>
<td>-2.06 (55.97)</td>
<td>38.16 (46.07)</td>
<td>2.39 (24.30)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normalized Code-Generation</td>
<td>$n = 22$</td>
<td>$n = 19$</td>
<td>$n = 19$</td>
<td>$n = 19$</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>12.96 (42.08)</td>
<td>31.67 (60.20)</td>
<td>9.74 (22.45)</td>
<td>39.56 (42.20)</td>
</tr>
</tbody>
</table>

Table 5.2: Mean and standard deviation for normalized gain scores (by percentage) for each question type

from pretest to posttest for high pretest scores). Conversely, in the experimental condition, the translation gain was positive. The opposite pattern occurred for the code-trace questions, with the control group gain positive and the experimental group staying more or less the same (average gain dipped just below 0 but the magnitude was small). For the code-generation question, both conditions had positive gains.

When we checked for outliers in normalized gain scores, several were flagged for each question type (see Figure 5.2). There was one outlier in the control conditions for translation question scores, four outliers in the experimental condition for code-trace question scores,
as well as outliers in both conditions of the code-generation question scores (three in the control condition and one in the experimental condition). Since outliers were detected, the descriptive analyses were re-run. The patterns between conditions for each question type remained the same (see Table 5.2, right).

Outliers aside, there are a few notables features to point out for the distribution of normalized gain scores (shown in Figure, 5.2). For the translation questions, the control group has more varied scores than experimental group based on the whisker to whisker range. In addition, the lower 50% of scores in control condition were more variable compared to the experimental condition, and were near to, or below 0. This indicates that many students in the control condition performed worse on the posttest than they did on the pretest. For the code-generation scores, in both conditions there is little variability in the lower 50% of the scores. This is may indicate that students in the lower 50% of normalized gain scores received almost identical scores on their pretest and posttest. This is relatively unsurprising however, due to the low raw score attributed to the code-generation.

An independent samples t-test on the normalized gain scores for each question type was run to determine the effect of condition on normalized gain. Since there were outliers, we conducted the analysis with and without the outliers. We begin for the results of the analysis with the outliers.

For translation questions, the experimental condition learned significantly more than the control condition, $t(31.2) = 2.35$, $p = .025$, $d = 0.71$). In contrast, for code-trace questions, the control condition learned significantly more than the experimental condition $t(39) = 2.54$, $p = .016$, $d = 0.79$. The difference in code-generation gain was not significant, $t(39) = 1.017$, $p = .282$, $d = 0.37$.

To check if outliers impacted the results, they were removed, and the analyses were re-run. The removal of the outliers did not impact the significance of the difference between
Figure 5.2: Distribution for normalized gain scores (%) by question type and condition
conditions for the translation and code-trace questions (translation: \( t(41) = 2.09, p = .043, d = 0.64 \); code-tracing \( t(33.31) = 3.07, p = .04, d = 0.92 \)). For the code-generation questions, the outliers were influential – when they were removed, the experimental condition learned significantly more the control condition, \( t(27.43) = 2.72, p = .011, d = 0.88 \).

In summary, while overall students did learn, a fine-grained analysis of scores broken down by question type revealed a more nuanced pattern. Translation activities given to the experimental group led to more learning of translation and code generation but led to less learning for code-tracing questions, as compared to the control group. The implications of these findings are discussed in the next chapter.
Chapter 6

Discussion

This thesis involved the design and evaluation of a computer tutor for code tracing. Two tutor versions were developed, one for the experimental condition in which students were asked to translate Python syntax to English during code-tracing activities, and one for the control condition in which students completed the code-trace activities without translation. There has already been some work on code-tracing tutors (Jennings & Muldner, 2020; Nelson et al., 2017). We based the design of the present tutor on the one used in Jennings and Muldner (2020). The novel aspect of the present work was the integration of the translation activities in the experimental version of the tutor.

We evaluated the impact of the two tutor versions on learning through a two-condition study using a between-subjects design. Learning was operationalized by normalized gain for three types of test questions: translation questions, code-tracing questions, and code-generation questions. These questions are associated with different skills needed for programming (Xie et al., 2019) and thus learning was reported on them separately. We begin by discussing the result for each question type.
6.1 Learning Results by Question Type

6.1.1 Translation

For the translation questions, students in the experimental condition learned significantly more than students in the control condition, with a moderate effect size. This result was expected because the experimental group practiced translation activities while the control group did not. Feedback may have had a role in fostering learning for the experimental group (Anderson et al., 1989; Shute, 2008).

Notably, the control group’s posttest scores were lower than the pretest scores, resulting in an average negative normalized gain score. Outliers were removed to verify this result, and the magnitude of the negative gain score was reduced but not eliminated. We originally considered the possibility of the pretest to posttest loss to be due to participant fatigue (i.e., by the posttest, participants may have been tired and not trying very hard). However, this is unlikely as it should have also affected the experimental group. While it is possible that the control group’s net loss may be due to measurement error, the most likely explanation is the gap between the lesson (that included translation information) and the posttest. This gap may have resulted in the control group participants forgetting translation concepts from the lesson. The experimental condition received support for translation with the tutor after the lesson, which may have reinforced concepts from the lesson (helping them learn) or made them less likely to forget concepts from the lesson (or some combination of both) in comparison to the control condition.

As far as the effect of self-explanation (in the form of translations), our results replicate prior research indicating that self-explanation improves learning (Conati & Vanlehn, 2000; Alevan & Koedinger, 2002; Fabic et al., 2019). Students, when prompted to self-explain (in the experimental condition) learned more on translation questions than students who were
not prompted to self-explain by translating.

### 6.1.2 Code Tracing

For the code-tracing questions, students in the control condition learned significantly more than students in the experimental condition, with a large effect size. This is contrary to the expected outcome and is not due to influential outliers – after they were removed, the significant effect remained along with a large effect size (based on Cohen’s guidelines (Cohen, 1988)). Previous research demonstrated that code-tracing instruction and practice leads to increased learning (Nelson et al., 2017; Lee & Muldner, 2020; Jennings & Mulder, 2020). The control group’s outcome supported this pattern (i.e., this group improved from pretest to posttest). However, the experimental group did not improve their code-tracing skills from the code-tracing tutor (after outliers were removed, the magnitude for the gain score was negligible – so this group essentially stayed at the same level of performance from pretest to posttest). There are a number of possible explanations for these results. Translation may have increased cognitive load, which can reduce learning outcomes (Sweller, 2011). Translations added an extra feature to the activity for the participant to pay attention to, and cognitive load theory predicts that splitting a student’s attention between features can hinder learning.

In the experimental condition, students had to switch between the translation activity and the code-tracing activity, splitting their attention between the two tasks.

Other factors to explain the code-tracing result include the timing of translations, the effort required for translation, and/or feedback. The timing of the translations may have not been ideal – perhaps if the tutor prompted the student for translations at a different time rather than concurrently during the code-trace activity, different results may have been obtained (this is expanded on in the future work section that follows). In contrast to entering values of variables into the code-trace table, writing translations requires more effort. This
increased effort may have led to fatigue, which in turn may have impacted the results of the experimental condition by causing students to perform worse on the posttest at the end of the study.

6.1.3 Code Generation

For the code-generation question, students in the experimental condition learned more than students in the control condition, with a large effect size. The code-generation question was included in the pretest and posttest as a transfer question. Code generation requires knowledge of algorithms, which are high-level recipes of how to solve programming problems. The translation group was scaffolded to generate explanations of the program lines beyond programming syntax. Since these explanations were in plain English, they likely corresponded to representations more meaningful than programming syntax, facilitating learning of basic code schemas (i.e., the algorithms), useful for the code-generation question. However, this is speculative and so further research is needed to investigate the relationship between the two question types.

As part of exploratory analysis, we checked for the relationship between code-tracing and code-generation gain scores using a Pearson’s correlation within each condition. There was a weak correlation between the normalized gain for code-tracing scores and code-generation scores in both conditions (the experimental group: $r(15) = 0.56, p = .019$; the control group: $r(20) = 0.48, p = .022$). Therefore, a student’s learning of code tracing correlated with their learning of code generation.
6.2 Limitations and Future Work

A limitation of the present work is that by chance, there were a priori differences between the two conditions based on pretest scores. Descriptively, the experimental group had higher scores for all three question types, however only one of the differences was significant (code-tracing scores). While using normalized gain aimed to account for these differences, it does not negate the possible effects on the results. It may be that higher prior-knowledge student has less knowledge to gain (this is what using normalized gain aims to address). On the other hand, more prior knowledge may advantage the experimental group as prior knowledge on tasks can boost performance on the posttest. However, there was little indication of this in the present study: the experimental group had significantly higher pretest scores than the control group, and yet the control condition learned significantly more than the experimental group on the code-tracing questions.

The design of the tutor needs work to maximize learning. The tutor required translations to be completed concurrently with a code-trace activity. However, this appears to have unintentionally hindered the learning of code-tracing skills (as described previously). In chapter 3, we described an alternative design of the tutor requiring students complete all the translations before completing the code-trace activity. Despite this alternative design’s possible limitation in terms of increasing cognitive load, the present design also had limitations. As suggested above, completing translations during a code-trace activity may have led to increased cognitive load due to frequent switching between two different tasks (translation, code tracing). The present design may therefore hinder learning due to increased cognitive load. In general, work is needed to identify a more effective tutor design as well as investigate the timing of the translation activities in relation to code-tracing activities.

In the present study, the experimental group did not receive immediate feedback for the translations. This timing for feedback may have been suboptimal for two reasons. First,
immediate feedback for translation would mirror the timing of feedback provided for code tracing in both conditions. Providing immediate feedback for translations requires natural language processing but this fell outside the scope of the project. Second, as Margulieux and Catrambone (2019) suggested, the scaffolding and feedback may have been excessive for students and may have negatively impacted their learning. In their study, students who received both scaffolding and feedback learned less than students who received only one form of support. There is increased scaffolding for the experimental condition in the present work (with scaffolding being provided for both tasks). This is another area for future work, specifically to investigate how levels of feedback for translations impact learning.

Related to the above point, natural language processing tools could provide another metric by which to assess learning. A classifier that uses natural language processing tools could assess the quality of translations that students generate. Prior research has suggested that students who produce higher-quality self-explanations learn more than students who produce lower-quality self-explanations (Chi et al., 1989). The classifier could be trained using supervised learning methods, in order to be able to assess the quality of the explanations that students. Further research could investigate the relationship between the quality of self-explanations generated and the learning of code-tracing skills.

6.2.1 Conclusion

The present thesis investigated the utility of self-explanation during code-tracing activities, by prompting students to translate lines of Python code into plain English explanations of what that line does. Two versions of a computer tutor were developed, one in which students only code traced (the control condition) and one in which students were prompted to complete translations while code-tracing. Learning was operationalized by normalized gain scores for three programming skills. Engaging in self-explanation through translations
while code tracing increased learning for translation skills and code-generation skills, but not for code-tracing skills.
References


57


Appendix A

Demographics Questions

The demographics questionnaire contained the following three questions:

- *What is your major?* (student writes an open-ended answer)

- *What is your gender?* (student selects one of Non-binary, Female, Male, or Other (open-ended answer))

- *How much prior programming experience do you have? Please select the most applicable answer.* Students selects on of the following:
  - None
  - 1 high school level course
  - More than 1 high school level course
  - 1 university level course
  - More than 1 university level course
Appendix B

Pretest

Question 1:
We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen.

```
1 outcome = 1
2 outcome = outcome + 10
3 print(outcome)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
What does the program from above (copied below) print to the screen?

Question 2:
We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen. Assume the user enters the name
“Ben”.

```python
1 greet = "Hi 
2 res = input("Name? ")
3 res = res + greet
4 print(res)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation: What does the program from above (copied below) print to the screen?

**Question 3:**

We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen. Assume the user enters the “Ontario”.

```python
1 answer = input("Enter a city ")
2 counter = 0
3 if answer == "Ottawa":
4     counter = counter + 1
5 print(counter)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation:
Line 5 explanation:
What does the program from above (copied below) print to the screen?
Question 4:
We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen.

```
1 counter = 1
2 while True:
3     if counter > 2:
4         break
5     print("hi ")
6     counter = counter + 1
7     print("value: ", counter)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation:
Line 5 explanation:
Line 6 explanation:
Line 7 explanation:

What does the program from above (copied below) print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.

Question 5:
What does the program below print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.
What does the program from above (copied below) print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.

**Question 6:**
Summarize what the program below does by providing a brief high-level explanation (e.g. what does the program achieve and how does it do so?). (2-3 sentences maximum).

```python
1 score = 2
2 stuff = "x"
3 while True:
4     answer = input(" letter? ")
5     if score < 0:
6         break
7     score = score - 1
8     stuff = stuff + answer
9     print(score)
10 print(stuff, score)
```

**Question 7:**
This is a code-generation transfer question, the only question on the test that asks student to write Python code.

Write a program that asks the user for their password – if they get it right in 3 tries or less, they get a message saying “Welcome”; if they get it wrong 3 times, they get a message saying “You are locked out”. Assume the correct password is “test13”.

63
Appendix C

Posttest

Questions on the posttest. Each question (except for question 7 was accompanied by a program). The pretest was administered online through Google Forms. **Question 1:**

We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen.

```
1  result = 5
2  result = result + 20
3  print( result )
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:

What does the program from above (copied below) print to the screen?

**Question 2:**

We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen. Assume the user enters the name
"Kiri".

```
greet = "Hello 
res = input("Name? ")
res = greet + res
print(res)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation: What does the program from above (copied below) print to the screen?

**Question 3:**

We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen. Assume the user enters the "Canada".

```
location = input("Enter a city ")
counter = 0
if location == "New York":
counter = counter + 1
print(counter)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation:
Line 5 explanation:
What does the program from above (copied below) print to the screen?
Question 4:

We have added line numbers to the program below. Provide an explanation for each line of code, and explain what the program prints to the screen.

```
1  tracker = 0
2  while True:
3      if tracker > 3:
4          break
5      print("hello ")
6      tracker = tracker + 2
7      print("goodbye ", tracker)
```

Line 1 explanation:
Line 2 explanation:
Line 3 explanation:
Line 4 explanation:
Line 5 explanation:
Line 6 explanation:
Line 7 explanation:
What does the program from above (copied below) print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.

Question 5:

What does the program below print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.

```
1  current = 7
2  tracker = 6
3  while True:
4      if current < 2:
5          break
6      tracker = tracker - 2
7      current = current - tracker
8      print(current, tracker)
```
What does the program from above (copied below) print to the screen? Show how you got your answer by showing the values of variables and how they changed as the program executed.

**Question 6:**
Summarize what the program below does by providing a brief high-level explanation (e.g. what does the program achieve and how does it do so?). (2-3 sentences maximum).

```python
value = 5
contents = "m"
while True:
    stuff = input(" letter? ")
    if value < 0:
        break
    value = value - 2
    contents = contents + stuff
print(value)
print(contents,value)
```

**Question 7:**
This is a code-generation transfer question, the only question on the test that asks student to write Python code.

Write a program that asks the user for the Capital of Canada – if they get it right in 3 tries or less, they get a message saying “Correct”; if they get it wrong 3 times, they get a message saying “Incorrect”. Assume the correct answer is “Ottawa”.
Appendix D

Consent Form

The consent form was administered through Google Forms. Three different versions were used, the only differences being the compensation/incentives changing slightly, as participants from either of the courses received a 2% bonus to their final grade for that course specifically, and participants not from the two courses were provided with a 25$ e-transfer.

Name and Contact Information of Researchers

Principal investigator: Kasia Muldner Kasia.Muldner@carleton.ca

Michael Caughey michaeljcaughey@cmail.carleton.ca

Project Title Learning from Educational Technologies (Online Tutors & Problem Solving)

Project Sponsor and Funder NSERC Discovery Grant (Fund 315025)

Carleton University Project Clearance Clearance #: (116832) Date of Clearance: January 30, 2022

Invitation You are invited to participate in our research study. This study involves working online on some programming activities with an educational technology. No prior background in programming is required. You are eligible to participate if you do not have any programming experience or very limited experience (no more than one university-level class). The
information in this form is intended to help you understand what we are asking of you so that you can decide whether you agree to participate in this study. Your participation in this study is voluntary, and a decision not to participate will not be used against you in any way. As you read this form, and decide whether to participate, please ask all the questions you might have, take whatever time you need, and consult with others as you wish.

**What is the purpose of the study?** The purpose of this study is to learn more about how people solve programming problems and what kind of educational technology design best supports that process, including learning from it.

**What will I be asked to do?** If you agree to take part in the study, we will ask you to complete a brief demographics questionnaire and to solve a set of programming problems. We will record your problem-solving actions in the interface, through the educational technology that will log your entries and/or Zoom (your face will never be recorded). In-session data, such as the chat transcript, will be stored locally on the researcher’s password-protected computer. Operation data, such as meeting and performance data, will be stored and protected by Zoom on servers located in Canada, but may be disclosed via a court order or data breach. We may ask you to explain your answer to one of the problems you solved and will audio record your response (your face will not be recorded). This recording is a condition of participating in the study.

Please note that the recording of problem-solving actions is a mandatory part of the data collection for this study, and so a condition of consenting to participate (as noted above your face will not be recorded). The recording data will NOT be shared or publicly used, nor will it have your name associated with it. The study will take no more than 2 hours to complete (the length of time varies a little by participant).

**Risks and Inconveniences** We do not anticipate any risks to participating in this study, beyond the fact you may feel some discomfort when you feel you don’t know how to accomplish
the programming activities. During the study session, you have the right to not answer any questions and/or end your participation in the study for any reason, by stating that you do not want to continue.

**Possible Benefits** You may not receive any direct benefit from your participation in this study. However, your participation may allow researchers to better understand how to design educational environments that help students learn how to program.

**Compensation/Incentives**

*SONA students (CGSC 1001)*

As a token of appreciation, you will receive 2% extra credit for CGSC 1001. If you withdraw before finishing the experiment, the amount of compensation will be prorated by time (e.g. if you withdraw halfway through the study, you will receive half the compensation).

*CGSC 1005*

As a token of appreciation, you will receive 2% extra credit for CGSC 1005. If you withdraw before finishing the experiment, the amount of compensation will be prorated by time (e.g. if you withdraw halfway through the study, you will receive half the compensation).

*External Participants*

As a token of appreciation, you will receive $25 via e-transfer. If you withdraw before finishing the experiment, the amount of compensation will be prorated by time (e.g. if you withdraw halfway through the study, you will receive half the compensation).

**No waiver of your rights** By signing this form, you are not waiving any rights or releasing the researchers from any liability.

**Withdrawing from the study** If you withdraw your consent during the course of the study, all information collected from you before your withdrawal will be discarded. Since we do not keep a master list linking your participant ID to your name, withdrawing after the study is not possible.
**Confidentiality** You will be assigned a participant ID code so that your identity will not be directly associated with the data you have provided. Thus, none of the study materials will have your name on them (we will use anonymous identifiers like P1 instead). However, if you are being granted course credit for taking part in the study, identifying information will be retained by the SONA system using a code until the course credit is granted.

All data, including coded information, will be kept in a password-protected file on a secure computer. Research data will only be accessible by the researchers and the research supervisor and will not have any identifiable information. We will password protect any research data that we store or transfer. Once the project is completed, research data will be kept and potentially used for other research projects on this same topic.

The results of this study may be published or presented at an academic conference or meeting, but the data will be presented so that it will not be possible to identify any participants.

**Data Retention** Your Zoom screen recording will be destroyed as soon as it is analyzed (as soon as possible after the study completion and no more than a year). As far as the other data, while the de-identified data will not be shared or made public, we do plan to archive the information for potential future use by Muldner’s research team on related projects and future analysis. Personal identifiers are never attached to the data. Participant contact information will not be kept.

**New information during the study** In the event that any changes could affect your decision to continue participating in this study, you will be promptly informed.

**Ethics review** This project was reviewed and cleared by the Carleton University Research Ethics Board B. If you have any ethical concerns with the study, please contact Carleton University Research Ethics Board (by phone at 613-520-2600 ext. 4085 for CUREB B or by email at ethics@carleton.ca. During Covid, the Research Ethics Staff are working from home.
without access to their Carleton phone extensions. Accordingly, until staff return to campus, please contact them by email.

Statement of consent – sign name

Participant Signature:

"I voluntarily agree to participate in this study."

(Participant click on either a "Yes" or "No" option on a list)

Date of Participation