1-13-2020

The Curious Case of Loops

Briana Baker Morrison
University of Nebraska at Omaha

Lauren Margulieux
Georgia State University

Adrienne Decker
SUNY University at Buffalo

Recommended Citation

Morrison, Briana Baker; Margulieux, Lauren; and Decker, Adrienne, "The Curious Case of Loops" (2020). *Learning Sciences Faculty Publications*. 37.
doi: https://doi.org/10.1080/08993408.2019.1707544

Briana B. Morrison [a]*, Lauren E. Margulieux[b] and Adrienne Decker[c]

[a]*Computer Science Department, University of Nebraska Omaha, Omaha, Nebraska, USA;* [b]*Department of Learning Sciences, Georgia State University, Atlanta, Georgia, USA;* [c]*Department of Engineering Education, University at Buffalo, Buffalo, New York, USA*

*** Briana B. Morrison, PhD**
Assistant Professor
Computer Science Department
College of IS&T
University of Nebraska at Omaha
6001 Dodge Street
Omaha, NE 68182
402-554-3429
bbmorrison@unomaha.edu

* corresponding author

The Curious Case of Loops

Background and Context: Subgoal labeled worked examples are effective for teaching computing concepts, but the research to date has been reported in a piecemeal fashion. This paper aggregates data from three studies, including data that has not been previously reported upon, to examine more holistically the effect of subgoal labeled worked examples across three student populations and across different instructional designs.
Objective: By aggregating the data, we provide more statistical and explanatory power for somewhat surprising yet replicable results. We discuss which results generalize across populations, focusing on a stable effect size to be expected when using subgoal labels in programming instruction.
Method: We use descriptive and inferential statistics to examine the data for the effect of subgoal labeled worked examples across different student populations and different classroom instructional designs. We specifically concentrate on the potential effect size across samples of the intervention for potential generalization.
Findings: Two groups of students learning how to write loops using subgoal labeled instructional materials perform better than the others. The better performing groups were the group that was given the subgoal labels with farther transfer between worked examples and practice problems and the group that constructed their own subgoal labels with nearer transfer between worked examples and practice problems, both with medium-large effect sizes.
Implications: For educators wishing to improve student learning using subgoal labeled materials should either provide students with subgoal labels while having them practice with a wide range of practice problems or allow students to generate their own subgoal labels and practice problems within similar contexts.

Keywords: worked example, subgoal label, experiment, CS1

Introduction

Subgoal labeled worked examples have been effective for teaching computing concepts, but the research to date has been reported in a piecemeal fashion. Pieces of three experiments using subgoal labeled worked examples for learning loop constructs have been reported in various conference proceedings (*Study 1, Study 2, Study 3*) (Morrison, Decker, & Margulieux, 2016; Morrison, Margulieux, Ericson, & Guzdial, 2016; Morrison, Margulieux, & Guzdial, 2015). The current paper aggregates these pieces and reports on new data from the experiments to examine more holistically the effect of subgoal labeled worked examples across three student populations and across different instructional designs. The different instructional designs include the first instance of testing student-generated subgoal labels and the first instance of testing differing amounts of transfer between worked examples and practice problems, in any discipline. By aggregating data from all three studies, including data that has not been reported before, we provide more statistical and explanatory power for somewhat surprising yet replicable results. We discuss which results generalize across populations, focusing on a stable effect size to be expected when using subgoal labels in programming instruction.

Literature Review / Background

This section reviews the current literature for subgoal learning along with some background in cognitive load theory to allow for framing the studies. We first present a common instructional design tool, *worked examples*, before presenting *cognitive load theory*, as the examples given to illustrate cognitive load involve worked examples. We then focus on subgoal label research (in worked examples) conducted within the computing discipline.

Worked Examples

Worked examples are a type of instructional material used to teach procedural problem-solving processes. Worked examples give learners concrete examples of the procedure being used to solve a problem, showing the explicit steps in the problem-solving process. Eiriksdottir and Catrambone (2011) argue that learning primarily from worked examples may result in better initial performance as the worked examples are more easily mapped to the problems to be solved. They further posit, however, that learning from worked examples is less likely to result in retention and transfer of knowledge than learning from more abstract instructions. When studying worked examples, learners tend to focus on incidental features rather than the fundamental features of the problem. This occurs because the incidental features are easier to grasp for novices as they do not yet have the necessary domain knowledge to recognize the fundamental features of the worked examples (Chi, Bassok, Lewis, Reimann, & Glaser, 1989). For example, when studying physics worked examples, learners are more likely to recognize that the example has a ramp than that the example uses Newton's second law (Chi et al., 1989). Therefore, while worked examples can improve initial performance, when learners focus on incidental features, they ineffectively organize and store information, leading to ineffective recall and transfer (Bransford, 2000).

Cognitive Load

Cognitive load can be defined as "the load imposed on an individual's working memory by a particular (learning) task" (van Gog & Paas, 2012). The cognitive load imposed on the learner can directly affect knowledge retention and performance scores. Cognitive Load Theory (CLT) is grounded in the human architecture of the brain, which has a limited capacity for working memory. All information is processed in working memory

before being stored in long term memory. If the total amount of processing required to learn exceeds the limited capacity of working memory, then learning is impaired (Plass, Moreno, & Brünken, 2010). Current thinking defines two different types of cognitive load on a student's working memory: intrinsic load and extraneous load (Kalyuga, 2011; Sweller, 2010; Sweller, van Merriënboer, & Paas, 1998; van Merriënboer & Sweller, 2005).

Intrinsic load is a combination of the innate difficulty of the material being learned combined with the learner's existing knowledge. For example, a conceptual understanding of a loop and the individual programming constructs to write a loop are intrinsic load for a problem that uses a loop. Extraneous cognitive load occurs when the learner is presented with information that does not directly contribute toward learning and is thus, extraneous. For example, while studying a worked example of a loop for calculating the average of a group of scores, the details of how a specific score is calculated is necessary for processing the worked example but not intrinsic to understanding how to solve a problem using a loop. Thus, the incidental details of worked examples are often extraneous. Working memory resources that are devoted to information that is relevant or germane to learning are referred to as 'germane resources' (Sweller, Ayres, & Kalyuga, 2011).

The intrinsic and extraneous loads may be moderated through careful design of the instructional materials. Intrinsic load should be managed so that learners are not given too much new information to process at once. While some extraneous load is inevitable, instructional materials should attempt to eliminate unnecessary extraneous load. Worked examples, when carefully designed, can accomplish both of these goals (Sweller et al., 1998).

Subgoal Labels

To guide learners' attention away from incidental details and promote deeper processing of worked examples for improved recall and transfer, the subgoal learning framework can be used to design worked examples that emphasizes problem-solving structure. The subgoal learning framework is a strategy used predominantly in STEM fields to help students deconstruct problem-solving procedures into subgoals, or the functional parts of the problem-solving procedure, to better recognize the fundamental components of the problem-solving process (Atkinson, Catrambone, & Merrill, 2003). Subgoals can be thought of as the building blocks of procedural problem solving and they exist for all problem-solving procedures except the simplest ones.

Subgoal labeling is a specific technique used to promote subgoal learning. It has been used to help learners recognize the fundamental structure of the problem-solving procedure being illustrated in a worked example (Catrambone, 1994, 1996, 1998). Subgoal labels are function-based instructional phrases that explain to the learner the purpose of that step, or subgoal, in the problem-solving process. In Figure 1, the first two lines of code have the subgoal label “Initialize Variables.” This label provides information about the purpose of that subgoal and the function behind the steps within it. Studies (Atkinson, 2002; Atkinson & Derry, 2000; Catrambone, 1994, 1996, 1998; Margulieux & Catrambone, 2014; Margulieux, Guzdial, & Catrambone, 2012) have consistently found that subgoal-oriented instructions improved problem-solving performance across a variety of STEM domains, such as programming (e.g., (Margulieux et al., 2012)) and statistics (e.g., (Catrambone, 1998)).

[Figure 1 goes about here.]

Giving subgoal labels in worked examples improves learner performance while solving novel problems without increasing the amount of time learners spend studying

instructions or working on problems (Margulieux et al., 2012). From a cognitive perspective, it is thought that subgoal labels are effective because they visually group the problem-solving steps within the worked examples into subgoals and give meaningful labels to the groups (Atkinson et al., 2003). This subgoal labeled format highlights the structure of the examples, helping students to focus on the structural features of the problem and allows the learner to more effectively organize the information (Atkinson, Derry, Renkl, & Wortham, 2000). Because learners are more focused on the structural features of the worked example allowing more effective organization of the information, subgoal labels may reduce the extraneous cognitive load that can hinder learning but is inherent in worked examples (Renkl & Atkinson, 2002).

Subgoal labels that are context independent are the most effective type of subgoal labels (Catrambone, 1995, 1998). Catrambone found that learners who were given abstract labels (e.g., Ω) and had sufficient prior knowledge performed better than those who were given context specific labels (e.g., initialize accumulation loop variables) on problem-solving tasks done after a week long delay or in problems that required using the problem-solving procedure differently than demonstrated in the examples (Catrambone, 1998). Catrambone explained this finding by arguing that learners with sufficient prior knowledge could correctly explain to themselves the purpose of the abstract subgoal and that they presumably had to self-explain due to the abstract nature of the label. He argued that the self-explanation was more effective than providing context specific labels.

Self-explanation

A common and effective type of constructive learning that might help learners

understand subgoals is self-explanation. Self-explanation is a learning strategy in which students use prior knowledge and logical reasoning to make sense of new information and gain knowledge. A review of self-explanation studies found it is effective across a range of domains if the domain has logical rules with few exceptions (Wylie & Chi, 2014).

Self-explanation of a worked example's solution identifies structural features and reasons about the function of the problem-solving steps (Bielaczyc, Pirolli, & Brown, 1995). The purpose of self-explanation is similar to that of subgoal learning. By self-explaining worked examples, learners are more likely to recognize structural versus superficial features. However, learners do not often engage in self-explanation without explicit prompting. Many studies (e.g., Chi et al., 1989) found that 10% or less of learners self-explained examples without external prompting. Most of the time learners can self-explain if they devote additional resources to the task (Wylie & Chi, 2014) and if they are reminded and guided to do so. Research has found little difference in the learning outcomes of students who self-explain on their own or are prompted to self-explain (e.g., Bielaczyc et al., 1995). This suggests that self-explanation itself is the cause of learning benefits.

Parsons Problems

Before describing how subgoals have been used in computer science education, we should explain a type of assessment used in this research, Parsons problems. When learning programming, students must learn a new language - the programming language used to communicate instructions to the computing agent - with its own unique syntax. This level of intrinsic cognitive load can overwhelm the learner, so researchers have sought ways to eliminate or reduce the learning of programming language syntax

(Resnick et al., 2009). For text-based programming languages, one way to assess student knowledge without requiring syntax knowledge is to use Parsons problems (Parsons & Haden, 2006). In Parsons problems, correct code is broken into code fragments that students then put into the correct order with the correct indentation. Parsons problems require a lower cognitive load on the learner because the search space is limited to only the code fragments in the problem and there is no possibility of syntax errors. Using Parsons problems for assessment of student knowledge allows students without syntax knowledge of the programming language to demonstrate procedural problem-solving knowledge.

Subgoals in Computer Science

Subgoal learning was first applied to programming education in the context of an experimental laboratory with psychology undergrads as participants. Due to this context, the programming procedure being taught had to be accessible to absolute novices. Thus, participants were taught to create apps in Android App Inventor. In this highly controlled environment, subgoal labeled worked examples were found to improve problem-solving performance by 8% (Margulieux, Guzdial, & Catrambone, 2012). From that experiment, research has focused on testing subgoal labeled worked examples in more authentic programming education environments, including online learning with K-12 teachers (Margulieux, Catrambone, & Guzdial, 2016), a game-based K-3 setting (Joentausta & Hellas, 2018), and in open educational resources that crowdsource subgoal labels (Kim, Miller, & Gajos, 2013). Our research applies subgoal learning to an introductory programming course, specifically to students who were learning to solve problems using `while` loops.

Our first study (*Study 1*) (Morrison et al., 2015) tested hypotheses related to

whether using subgoal labels to teach `while` loops would produce results similar to those achieved in other disciplines. Learning to use `while` loops is cognitively demanding, and the study proposed that using subgoal labels to help students learn would reduce the cognitive load imposed during learning. Because students were several weeks into an introductory programming course, we also recognized that they would have some prior knowledge that was relevant to solving the loop problems. For this reason, we hypothesized that students might better learn the subgoals of the procedure if they were prompted to self-explain the subgoals, rather than being given subgoal labels that were already defined. Self-explaining the subgoals, if students were able to do it, would encourage active learning of the subgoals and lead to deeper learning than viewing existing subgoal labels, which would lead to passive learning.

To test this hypothesis, the study divided the participants into three treatment groups, each with its own instructional materials: learning with no subgoal labels (*No Subgoal*), learning with given pre-defined subgoal labels (*Given*), and asking participants to generate their own subgoal labels after some initial training (*Generate*). Each treatment group was then subdivided into two sections: isomorphic (near) or contextual (far) transfer between worked examples and practice problems (see Method-Design for more information on transfer). Like self-explaining subgoals, contextual (far) transfer between worked examples and practice problems was expected to promote deeper learning and improve later problem-solving performance, if students could successfully engage in it. Contextual transfer was also expected to be highly cognitively demanding and perhaps unachievable for many students.

This first study found that students who learned with subgoal labels (either given or generated) performed better on the code writing assessments than those who learned

without subgoal labels. Within the given and generated groups, the best performing group depended on the type of transfer between worked example and practice problems that they received.

The unexpected results occurred with the given subgoal label group. Cognitive Load Theory predicts that learning with given subgoal labels and no contextual transfer should impose lower cognitive processing than learning with given subgoal labels and contextual transfer and thus result in better learning. The contextual transfer would require additional working memory to process, reducing learning. However, the results from the first study directly contradict this prediction. *Study 1* found, unlike the other two treatment groups, that participants who learned with given subgoal labels and contextual transfer significantly outperformed the given subgoal labels with isomorphic problems, completely opposite from what Cognitive Load Theory predicts. We examined whether this main finding was an anomaly or if could be replicated.

In a follow up paper (*Study 1 follow-up)* (Morrison, Margulieux, et al., 2016), we examined the performance of students on a Parsons problem assessment, after having learned loop problem solving in one of the treatment groups (with no subgoal labels, with given subgoal labels, or generating their own subgoal labels). We found that students who were given subgoals performed statistically significantly better than those who had no subgoals or who generated their own subgoals, regardless of transfer condition. Participants that were given subgoal labels performed overall better than those that did not have subgoal labels and those that generated their own subgoal labels. Though participants in the generate labels and no labels conditions performed equally, participants who generated their own labels completed the task faster than those who did not receive labels.

In *Study 2* (Margulieux, Morrison, Catrambone, & Guzdial, 2016), the examination of the quality of the learner-generated labels from a new population of students and how this affected problem-solving performance was reported. *Study 2* found that twice as many participants generated specific labels than general labels, but a larger percentage of participants who received contextual transfer generated general labels than those who had isomorphic transfer. Participants who learned with isomorphic transfer and generated their own labels performed relatively well, regardless of the specificity of their labels. For those that learned with contextual transfer, their performance depended on whether they created specific or general labels. Those who created specific labels performed as poorly as the worst performing group, those who received no subgoal labels with contextual transfer. On the other hand, participants who created general labels with contextual transfer performed better than any other group.

The *Study 3* (Morrison, Decker, et al., 2016) paper replicated *Study 1* (Morrison et al., 2015) with a third population of students. The results supported the findings from the previous studies: participants who learn by generating subgoal labels (using isomorphic worked example – practice problem pairs) performed the best, and statistically better than if they had been worked example – practice problem pairs with contextual transfer. Despite the previous publications that report results of each of the three experiments individually, we have yet to report all of the data from these three experiments or examine them holistically to determine the cross-population effects of the subgoal labeled worked examples. This paper addresses this gap.

Present Study

In this paper we examine the effect of learning subgoals through different instructional methods (i.e., given labels versus generated labels compared to unlabeled)

and transfer distance between worked examples and practice problems (i.e., isomorphic or contextual transfer) across three separate, but comparable, populations. This new analysis of the data allows us to report findings that were excluded from previous conference proceedings and explore the average effect of the interventions to determine a stable effect size across populations. We have the following research questions:

RQ1: How do different instructional methods of learning with subgoals (either given or learner generated) affect problem-solving performance?

RQ2: How does transfer distance (i.e., isomorphic transfer (changing the values in a problem with the same context) or contextual transfer (changing the context, or cover story)) from worked examples to paired practice problems affect problem-solving performance?

To measure performance, we used three different assessments: 1) four novel coding writing problems, 2) one Parsons problem, and 3) a post test of five multiple choice questions. The three assessments were chosen to represent three levels of difficulty and application of knowledge. Code writing was intended to be the most difficult and required students to recall the problem-solving process from memory. The Parsons problem was intended to assess knowledge of the problem-solving process while allowing students to recognize, rather than recall, the procedure. Furthermore, students do not have to determine how to apply a conceptual understanding to a new context in Parsons problems because the lines of code are provided for them. Therefore, increasing the transfer distance between worked examples and practice problems might not necessarily improve Parsons problem performance, though it was expected to improve code writing performance. The multiple-choice questions required students to trace the code and determine which answers containing possible outputs were correct.

These questions were intended to be the easiest questions and a learning check to identify participants who were not engaging in the instruction. Additionally, we measured cognitive load related to the instructional materials using the (Morrison, Dorn, & Guzdial, 2014) instrument and time on task for both the learning period and each assessment.

Method

Design

The experiment had two manipulations: the format of worked examples and the transfer distance between worked examples and practice problems. The worked example either had no subgoal labels (i.e., *No Subgoal*), had subgoal labels created by experts (i.e., *Given*), or included a placeholder for the participant to fill in their own subgoal label (i.e., *Generated*). In the *No Subgoal* condition (Control group A in the Appendix), the worked example is presented in a step by step solution of how to develop the code solution for the problem, including code comments. In the *Given* condition (Subgoal given, group B in the Appendix), the worked example is the same but broken into groups and labeled by the subgoal associated with the task. One subgoal may include more than one step. Code comments were identical to the control condition. For the *Generated* condition (Subgoal generate group C in the Appendix), the worked example was broken into groups, as in the Given condition, but instead of including the expert created subgoal label, a blank space was included to allow the participant to type in their own subgoal explaining what the pieces of code accomplished.

The second manipulation involved the differences between the worked example and practice problem given to the students. As can be seen in the Appendix (worked examples compared to practice problems), for the isomorphic (near) transfer problems

the context of the problem for the worked example and the practice problem is identical, and only the values being manipulated change. For the contextual (far) transfer problems, the context of the worked example and the practice problem are different, however the solution has an identical format. The experiment measured performance with pre- and post-tests, problem-solving tasks (both writing code and completing a 13-step Parsons problem), self-reported cognitive load on the (Morrison et al., 2014) instrument, and time on task.

Instructional Materials

In this study, we developed instructional materials to teach introductory programming students to solve programming problems using `while` loops. We selected the topic of writing indefinite loops for several reasons: 1) based on experience we know that students can struggle with the introduction of repetition statements, 2) `while` loops are the most general form of a repetition control structure allowing any type of loop to be written, and 3) teaching of this topic occurs in the early part of the term allowing us to reach the maximum number of students – typically before the withdrawal date for the term passed.

The materials used pseudocode so that students from multiple universities and courses that used different programming languages could participate. Pseudocode is easy for students to understand regardless of the programming language that they are learning (Tew & Guzdial, 2011). The first two experiments started before students had learned to use `while` loops in their courses, and the third experiment was conducted after students had been introduced to `while` loops. The procedure took about two hours to complete. In most cases, the experimenters conducted the experiment in a regularly scheduled lab for the programming courses from which they recruited

participants. The labs were held in closed classrooms with at least one computer per student. Some participants completed the procedure as an at-home assignment.

The instructional materials were three separate worked examples interleaved with a practice problem after each worked example. The format of the worked examples can be seen in the Appendix (Worked Examples). Each worked example appeared on one screen, followed by the practice problem on the next screen. Students could go back and forth between the worked example and the practice problem during the instructional period. Once the student reached the assessment portion of the study they could not go back to the instructional materials.

At the beginning of the session, the experimenters introduced the study explaining that they would learn to solve problems using `while` loops and that the materials they received would help them to achieve this. The experimenters then gave students a link to a SurveyMonkey survey where all of the materials and assessments were hosted. Participants worked independently and could ask for help from the experimenter on administrative tasks (e.g., "What is my participant number?") but not for help on the programming tasks (e.g., "How do I increase the loop control variable?"). Because students worked independently, some completed the tasks faster than others, and SurveyMonkey recorded how quickly each student progressed through the various stages of the experiment.

Assessments

After completing the instructional period with worked examples and practice problems, participants were asked to solve four novel problems using `while` loops. All the assessment problems required contextual transfer from the worked examples and practice problems that participants used to learn the procedure. No subgoal labels

appeared in any of the assessment problems. We scored participants' problem-solving solutions to create a problem-solving score. We evaluated the solutions line-by-line rather than as a whole to provide more sensitivity in the score. Each correct line of code earned one point for a maximum score of 44 points across four questions. Lines of code were considered correct if they were conceptually correct, regardless of typos or syntax errors. Logic errors (e.g., having < rather an <=) made the line incorrect. We decided to score for conceptual and logical accuracy rather than absolute accuracy because the participants were inexperienced programmers.

We also measured participants' problem-solving procedural knowledge with a Parsons problem. We scored participants' Parsons problem answers for correct order. Because the Parsons problem had 13 pieces of code to rank order, the maximum score was 13. Participants earned one point for each piece of code that was in the correct order relative to the piece before it. For example, if a participant's solution ranked the 6th, 7th, and 8th pieces of code in the 7th, 8th, and 9th positions, they would lose only the first point because it did not follow the 5^{th} piece of code. The 7^{th} and 8^{th} pieces would still be in order, relative to other pieces of code, and counted as correct. This scoring scheme was considered better than scoring for exact order because it does not penalize later pieces of code for earlier mistakes.

Procedure

Most participants completed the experiment during one of their lab sessions in a computer laboratory. Students had an option to complete an alternative assignment, but none selected that option. Participants worked independently, and each session included between 15 and 30 people. The sessions typically lasted between 1 and 1.5 hours, depending on the rate at which participants completed the tasks. For students in the lab

setting, a few stragglers were asked to leave at the end of 2 hours due to the next class arriving.

First, participants completed a demographic questionnaire and the pre-test. Next, they began the instructional period. The instructional period began with training. Participants who were going to generate their own subgoal labels received training to create subgoal labels (see the Appendix – How to Make Subgoal Labels). The training included instructions about creating subgoal labels, examples of a subgoal labeled worked example, and activities to practice creating subgoal labels on simple algebra problems designed be easy for any college student so that they could focus on creating labels. Participants who did not generate their own subgoal labels received training to complete verbal analogies (available in the Appendix – Verbal Analogies). Verbal analogies (e.g., water : thirst :: food : hunger) were considered a comparable task to subgoal label training because they both require analyzing text to determine an underlying structure. Like the subgoal label training, the analogy training included instructions, worked examples, and activities to practice.

Following the training, the instructional period provided worked examples and practice problems to help participants learn to use `while` loops to solve problems. Once participants completed the instructional period, they started the assessment period. Throughout the procedure, the time taken to complete each task was recorded. A diagram of the entire study procedure can be seen in Figure 2.

[Figure 2 goes about here]

Participants

Participants across the three experiments were 220 students recruited through programming courses and offered course credit for completing a lab activity as

compensation. To account for possible effects of prior experience, participants reported whether they had experience with programming and/or using loops during high school (AP courses or otherwise) and college. Other learner characteristics that participants provided were gender, age, academic major, high school grade point average (GPA), college GPA, whether English was their primary language, number of years in college, self-reported comfort with computers, expected difficulty of completing the programming task, and prior courses in programming. Participants were randomly assigned to intervention conditions to avoid possible confounds caused by learner characteristics. To ensure that there were no confounds, learner characteristics and problem-solving performance were correlated using Pearson's r for continuous learner variables and Spearman's ρ for dichotomous learner variables. The results of these analyses are reported in Tables 1, 2, and 3.

In addition to asking students about their prior experiences with programming and using loops, participants completed a pre-test to measure their prior knowledge of solving problems using `while` loops. The pre-test included five multiple-choice questions from AP CS A exams. Participants who answered more than two questions on the pre-test correctly were excluded from analysis to reduce potential error because the instructional materials were intended for novices. Participants who did not complete all components of the experiment were also excluded from analysis. The numbers of students excluded were relatively low and detailed in the following sections.

Experiment 1 Participants

Participants were 66 students from one of four introductory programming courses at a technical university in the southeast United States. The experiment occurred before students learned about loops in their courses. Students performed poorly on the pre-test,

M = 1.2 out of 5 points, and 32% of participants earned no points. Six students (out of 72, 8%) were excluded from analysis for high pre-test scores. No statistically significant relationships between all assessments and learner characteristics were found for most variables. Comfort with computers, expected difficulty of task, and taking a prior course, however, correlated with problem-solving performance. To ensure that no conditions had an advantage over the others based on these learner variables, we inspected the means for each of these learner variables within each condition. We found no meaningful differences (i.e., more than a few decimal points) among conditions.

[Table 1 goes about here]

Experiment 2 Participants

Participants were 54 students from introductory programming courses at a different technical university in the southeast United States. Unlike in Experiment 1, only 23% of participants were computer science majors. The majority of students were taking a Computational Media course. Many of them were likely taking the course because the university requires that all students take a programming course, and this course is designed specifically for students not majoring in computing. This sample characteristic explains the relatively high average age and number of years in college for participants.

The average score on the pre-test was low, *M* = 1.6 out of 5, and 23% of students earned no points. Five students (out of 59, 8%) were excluded from analysis for high pre-test scores. The only learner variable that correlated with assessment scores was high school GPA. The mean high school GPA for each experimental group was inspected to ensure that no groups had an advantage over the others. Each mean was within a few decimal points of the others.

[Table 2 goes about here]

Experiment 3 Participants

The last site used to collect data was a technical university in the northeast United States. The final experiment had a larger number of participants than the first two, 100 students. The final experiment also included students from first-semester introductory programming courses, like the first two experiments, and students in a second-year course. Collecting data from both the first-semester and second-year course in the computing curriculum allowed us to explore how prior knowledge impacted the results because the students in the second-year course would have already learned, practiced, and been tested on solving problems with `while` loops in a previous course (*Study 3*) (Morrison, Decker, et al., 2016).

In this experiment, both first-semester and second-semester students had already learned to use `while` loops. To account for prior knowledge, participants completed the same pre-test as Experiment 1 and 2. The average score was $M = 2.3$ out of 5. Participants were not excluded from analyses based on their pre-test scores, unlike in the previous two experiments. At the University for this study, students are not given credit for AP CS courses. This led to a large number of students in the first-semester course having prior programming knowledge. If we had excluded students based on their pre-test scores, there would not have been enough statistical power in the analyses. Additionally, this manuscript aggregates the effect of subgoal labels across different populations; having more knowledgeable students represents a unique population compared to the first two studies. As in Experiment 1, comfort with computers, expected difficulty of task, and taking a prior course correlated with problem-solving performance. We again inspected the means for each of these learner variables within each condition to ensure that no condition had an inherent advantage over the others. No meaningful differences were found among conditions.

[Table 3 goes about here.]

Results

The data used for this paper have been partially reported in previous papers as independent experiments. The problem solving, post-test, and time on task data for Experiment 1 were published in (*Study 1*) (Morrison et al., 2015). The Parsons problem data for Experiment 1 were published in (*Study 1 follow-up*) (Morrison, Margulieux, et al., 2016). For both Experiment 1 and 2, the problem solving, Parsons problem, quality of generated labels, and time on task were published in (*Study 2*) (Margulieux, Morrison, Catrambone, & Guzdial, 2016). For Experiment 3, the problem solving and time on task data were published in (*Study 3*) (Morrison, Decker, et al., 2016). For some of the analyses reported in these papers, the differences among groups had meaningful effect sizes but were not statistically significant. By analyzing the data together, the sample size, and thus statistical power, will be large enough to produce reliable effect sizes and, if the differences are large enough, statistical significance.

In addition to adding statistical power to our analyses, this paper will include data that has not been reported before due to space constraints. The new data included in this analysis are cognitive load data for all three experiments, Parsons problem data from Experiment 3, and post-test data for Experiments 2 and 3.

For all dependent variables (i.e., problem-solving performance, post-test, Parsons problem, cognitive load, and time on task), we analyzed the distribution of scores for skewness and kurtosis to ensure normal distribution and, therefore, that parametric statistical tests, such as ANOVA, were appropriate. In addition, we visually inspected the histograms of scores for each measurement. In all cases, the skewness and kurtosis were within normal bounds (i.e., between -2 and 2 (Gravetter & Wallnau,

2016)) and histograms followed a normal distribution. Therefore, no outliers were excluded from analyses, and parametric tests are appropriate for analyses of the measurements.

Performance data

For our inferential statistics, we report two types of effect sizes. The first, est. ω^2, is for only omnibus analyses (i.e., ANOVAs) and describes how much of the variation in scores can be attributed to the manipulation (i.e., proportion of variance accounted for, PVAT). For example, for the problem-solving tasks, an est. ω^2 of .06 means that 6% of the variation in performance can be attributed to the instructional manipulations. In the social sciences, an est. ω^2 of .06 is considered a medium-sized effect (Cohen, 1969). The second effect size, *f* or *d*, was used for only our *post hoc* analyses to describe the difference between groups using the standard deviation as the unit of measurement. For example, for the problem-solving tasks, a *d* of .5 would mean that the difference between the means of two groups is half of the standard deviation for those groups. The statistic *d* is used for t-tests, and the statistic *f* is used for ANOVAs and is equal to 2*d* (Cohen, 1988). For example, an *f* of .25 is equal to a *d* of .5, and both indicate that the difference between means is half of a standard deviation, which is considered a medium-sized effect (Cohen, 1969).

Problem-solving score

The main dependent variable, score on problem-solving tasks, had a maximum score of 44. The overall mean score was 26.58, and the standard deviation was 14.05. For the omnibus ANOVA analyses of these data, worked example format and transfer distance were treated as randomly assigned variables. In addition, university, which was different for each experiment, was treated at a quasi-experimental variable. This nested

design allows us to combine the data from the three experiments while still accounting for possible differences among universities.

Problem-solving score depended on the interaction of the worked example format and transfer distance, F (2, 188) = 5.23, p = .028, est. ω^2 = .08 (see Figure 3), matching previous results from independent experiments (*Study 1, Study 2, Study 3*) (Margulieux, Morrison, et al., 2016; Morrison, Decker, et al., 2016; Morrison et al., 2015). Due to the interaction, the main effects of worked example format and transfer distance will not be reported to avoid confusion in interpretation (Maxwell & Delaney, 2004). Instead pairwise comparisons will be used as post hoc tests to explore the pattern of results. Exploring the effect of university, there was no interaction of university and worked example format, p = .37, university and transfer distance, p = .65, nor university, worked example format, and transfer distance, p = .20. In addition, there was no main effect of university, p = .12; therefore, the combined data from all three universities were used for the *post hoc* tests.

[Figure 3 goes about here]

For *post hoc* analysis, we used simple main effects. Simple main effects analyze the effect of one independent variable for each level of the other independent variables. For example, simple main effects analysis will explore the effect of worked example format twice, once within isomorphic transfer and once within contextual transfer. Because worked example format had three levels, the effect is analyzed with pairwise comparisons among each of the levels. Full results can be found in Table 4. The only two comparisons that were statistically significant were those within isomorphic transfer between given labels and generated labels and within contextual transfer between no labels and given labels. These results suggest that there are two levels of

performance, low and high. The two lowest scoring groups performed statistically worse than the two highest scoring groups (see Figure 3). The two groups in the middle did not perform statistically different than the others, but they are numerically close to the lowest scoring groups and had higher mean differences and effect sizes from the highest scoring groups. Thus, we consider the two middle groups as low performance groups.

[Table 4 goes about here]

To further explore performance, we split the problem-solving tasks into nearer (i.e., switched context) and farther (i.e., deviate from exact procedural steps) transfer from the instructional tasks. Switched context meant that we used the same type of contextual transfer as we used between the worked example and practice problem pairs. In this case, it describes transfer between the instructional tasks (i.e., worked example and practice problems) and the problem-solving tasks in this assessment. Procedure transfer means that the procedure used to solve the problem-solving task did not follow the exact same steps as the instructional tasks. For example, in the instructional tasks, participants had to use a `while` loop to find an average of a list, and in the problem-solving tasks, participants had to use a `while` loop to find an average of values that exceeded a threshold (examples can be found in the Appendix – Worked Example #1 and Assessment #2). The problem-solving task had extra steps but still used the same abstract procedure that was taught.

The results did not change when comparing groups within only the nearer or farther transfer tasks. In both cases, there was still a statistically significant interaction with the same pattern of scores, F (nearer; 2, 188) = 4.04, p = .02, est. ω^2 = .06, F (farther, 2, 188) = 2.99, p = .03, est. ω^2 = .05. These results suggest that the

interventions had the same effect on problem-solving performance regardless of the type of transfer that was required to complete the problem-solving tasks.

Parsons problem score

The Parsons problem score was based on one Parsons problem and had a maximum score of 13 for putting each of the lines of code in the correct order. The overall mean score was 6.20, and the standard deviation was 4.27. Like for problem-solving performance, in the omnibus ANOVA analyses of these data, worked example format and transfer distance were treated as randomly assigned variables and university was treated at a quasi-experimental variable.

Parsons problem score did not have a statistically significant main effect of worked example format, F (2, 188) = 1.11, p = .41, est. ω^2 = .03, transfer distance, F (2, 188) = 0.15, p = .73, est. ω^2 = .01, nor interaction of the worked example format and transfer distance, F (2, 188) = 1.50, p = .31, est. ω^2 = .03. These results align with results in (*Study 3*) (Morrison, Decker, et al., 2016) but not with (*Study 2*) (Margulieux et al., 2016), which found a main effect of worked example format and concluded that giving subgoal labels, regardless of transfer distance, improved Parsons problem score. This difference in results might be due to including only one Parsons problem in our protocol, possibly contributing to an unreliable measurement of Parsons problem performance. Based on the larger sample size of both the current analysis and that conducted in (*Study 3*) (Morrison, Decker, et al., 2016), we would expect that the current result is more reliable. Therefore, we would not conclude that giving learners subgoals labels necessarily results in better performance on Parsons problems after receiving instructional materials similar to ours.

In the current analysis, we found a main effect of university, F (2, 188) = 10.16,

$p = .04$, est. $\omega^2 = .06$. There was no interaction of university and worked example format, $p = .11$, university and transfer, $p = .51$, nor university, worked example format, and transfer, $p = .22$. The difference between University 1 ($M = 3.7$) and University 2 ($M = 4.6$) was not statistically significant, $t\ (116) = 1.35$, $p = .18$, $d = .25$. In contrast, University 1 performed much worse than University 3 ($M = 8.8$), $t\ (181) = 10.44$, $p < .001$, $d = 1.57$. Similarly, University 2 performed much worse than University 3, $t\ (133) = 5.53$, $p < .001$, $d = 1.04$. These results are not unexpected, though, given that the participants from University 3 had already learned about solving problems with loops in their programming courses. It is interesting that participants from University 3 performed statistically better than those in the other universities on the Parsons problem but not on the problem-solving tasks, which were writing code tasks. This supports the notion that students may demonstrate problem solving knowledge in Parsons problems even if they cannot in traditional code writing problems.

Post-test score

The post-test asked participants to complete, after instruction, the same five multiple-choice questions from the AP CS exam that they had completed prior to instruction. The maximum score was 5, and the mean was low, 2.40, with a standard deviation of 1.45. The post-test score did not have a statistically significant main effect of worked example format, $F\ (2, 188) = 1.37$, $p = .34$, est. $\omega^2 = .03$, transfer distance, $F\ (2, 188) = 0.24$, $p = .65$, est. $\omega^2 = .01$, nor interaction of the worked example format and transfer distance, $F\ (2, 188) = 1.39$, $p = .33$, est. $\omega^2 = .02$. These results align with individual experiment results from (*Study 1, Study 3*) (Morrison, Decker, et al., 2016; Morrison et al., 2015). In addition, there was no main effect of university, $p = .76$, interaction of university and worked example format, $p = .50$, university and transfer distance, $p = .85$,

nor university, worked example format, and transfer distance, $p = .27$. We would expect, based on the results of the problem-solving tasks and Parsons problem, that participants would score higher on this post-test. Moreover, we would expect that participants from University 3 would perform better on this test than other participants because they were not excluded from analysis due to high pre-test scores and because they had learned about loops in their course already. Therefore, we conclude that this post-test, perhaps because it measured code tracing skill more than problem-solving skill, did not effectively measure performance for any of the groups of participants, and we do not include this assessment when considering the conclusions of the study. These results support the idea that code tracing is a skill separate from code writing (Harrington & Cheng, 2018; Kumar, 2015).

Process data

To supplement our data about performance outcomes, we collected information about the learning process to explore differences among groups. These data include perceived cognitive load during instruction and time on task during instruction and assessment.

Cognitive load

The cognitive load survey asked participants questions about their cognitive load directly after instruction to measure their perceptions of cognitive load during instruction (Morrison et al., 2014). Each of the 10 questions asked participants to rate their perceived cognitive load (e.g., "The topics covered in the activity were very complex") on a scale from "0 – not at all the case" to "10 – completely the case," making the maximum score 100. The mean was 40.9 with a standard deviation of 14.6. Cognitive load did not have a statistically significant main effect of worked example format, F (2, 188) = .51, $p = .63$, est. $\omega^2 = .01$, transfer distance, F (2, 188) = 0.89, $p =$

.43, est. $\omega^2 = .02$, nor interaction of the worked example format and transfer distance, F (2, 188) = .56, $p = .60$, est. $\omega^2 = .01$. Furthermore, there was no main effect of university, $p = .35$, interaction of university and worked example format, $p = .51$, university and transfer distance, $p = .61$, nor university, worked example format, and transfer distance, $p = .20$, suggesting no differences among universities.

These results were not previously reported for individual experiments due to space constraints. In this case, though, finding no statistical difference is good as it suggests that students did not perceive a meaningful difference in mental workload even though the instructions asked them to engage in different tasks. One possible explanation of these null results is that participants in all conditions used the same amount of mental resources, whether they were engaging in our prescribed learning strategy or not. We have no supplemental evidence to make a strong argument for this possibility. We can say, however, that some participants performed better than others without perceiving differences in mental workload.

Time on task

The total amount of time that participants spent on the experiment was recorded. This includes time spent studying worked examples, solving practice problems, and completing the assessments. The amount of time that participants spent on the task depended on worked example format, F (2, 188) = 8.67, $p < .001$, est. $\omega^2 = .09$. There was no effect of transfer distance, F (2, 188) = 0.55, $p = .46$, est. $\omega^2 = .003$, nor was there an interaction, F (2, 188) = 1.20, $p = .30$, est. $\omega^2 = .01$. Performance did not interact with university either, F (2, 188) = 0.63, $p = .67$, est. $\omega^2 = .002$.

To explore the effect of worked example format on time on task, we used simple main effects analysis. Within the isomorphic transfer condition, *No Subgoal* participants

completed the task faster (*M* = 52 minutes, *SD* = 21 minutes) than participants in the *Given* (*M* = 72, *SD* = 27) or *Generate* (*M* = 71, *SD* = 29) conditions, Mean Difference = 20.1 and 18.8 minutes, *p* = .003 and .007, *d* = .83 and .75, respectively. The *Given* and *Generate* conditions did not differ on time on task, Mean Difference = 1.4 minutes, *p* = .85, *d* = .04. When considering the effect on time, it is important to remember that within the isomorphic transfer condition, participants who generated their own subgoal labels performed best, and participants without subgoal labels or who were given subgoal labels did not perform differently. This combination of results means that participants who generated subgoal labels with isomorphic transfer took longer than those who did not receive subgoals, but they performed better. In contrast, participants who were given subgoal labels with isomorphic transfer took longer than those who did not receive subgoals but did not perform better. Therefore, taking longer to complete the task did not result in better performance for each group.

Following a similar pattern within the context transfer condition, the *No Subgoal* participants completed the task faster (*M* = 59 minutes, *SD* = 25 minutes) than participants in the *Given* (*M* = 67, *SD* = 25) or *Generate* (*M* = 79, *SD* = 35) conditions, Mean Difference = 12.2 and 20.1 minutes, *p* = .076 and .005, *d* = .32 and .66, respectively. Though the difference between the *No Subgoal* and *Given* groups is not statistically significant, we argue that it is meaningfully significant, albeit small, based on the mean difference and *d* value. The *Given* and *Generate* conditions did not meaningfully differ on time on task, Mean Difference = 7.9 minutes, *p* = .22, *d* = .21.

A piece of information to highlight from these results is that the standard deviation for the group who generated subgoals with contextual transfer was 35 minutes, which is approximately 10 minutes more than the other groups. This means

that participants in this condition had much more variance in the amount time on task than those in other conditions. If we were to offer a *post hoc* explanation of this finding based on our observations as experimenters and exploring the data, we might argue that participants in this group were more likely to flounder and take an excessively long time to complete the experiment. This group had twice as many people as any other group who took 100 minutes or longer (6 participants compared to 1-3 participants in the other groups).

Similar to the isomorphic transfer condition, it is important to recognize that within the contextual transfer condition, participants who were given labels performed better than others. This combination of results means that participants who were given subgoal labels with context transfer took slightly longer than those who did not receive subgoals, but they performed better. Moreover, participants who generated subgoals with context transfer took substantially longer than those who did not receive subgoals, but they did not perform better. The combined results suggest that depending on the transfer distance and worked example format, better performance required more time on task, but more time on task did not guarantee better performance.

To explore the relationship between time on task and performance more deeply, we examine the correlation of these two dependent variables within each group. Overall, there was a strong, positive relationship between performance and time on task, $r = 0.43$, $p < .001$, as is typical in education research. However, this relationship was not consistent within each experimental group (see Table 5), suggesting that spending longer on the task did not necessarily coincide with higher performance. The relationship between time on task and performance was strongest when students learned subgoals with isomorphic transfer between examples and practice problems or when students did not learn subgoals with contextual transfer. The relationship was weakest

when students generated subgoals with contextual transfer or when students did not learn subgoals with isomorphic transfer. Therefore, despite the extra time that students learning subgoals spent on the task, their extra effort did not consistently result in higher performance. As such, we conclude that the benefit of learning subgoals (under particular circumstances) is due to more than coaxing students to spend more time on task.

[Table 5 goes about here]

Discussion

In the cumulative analysis of three studies that used the same experimental protocol across three groups of learners at different institutions, we found that the most effective instructional design interventions were those that 1) gave subgoal labeled worked examples with farther transfer between worked examples and practice problems or 2) asked students to generate subgoal labels for worked examples with nearer transfer between worked examples and practice problems. In our experiment, these two conditions performed equally, but in practice, there might be reasons to pick one over the other based on several factors, such as characteristics of students in the class, the teaching style of the instructor, or the instructional materials (e.g., curriculum or textbook) being used.

The students in the class might affect whether they will successfully generate subgoal labels. If the students are already engaging in self-explanation (e.g., they answer challenging questions in class), learn concepts quickly, or are highly motivated to learn the content, then promoting self-explanation through generation of subgoal labels might be particularly effective. When we analysed the content of the subgoals generated by students, we found that students who learned with contextual transfer and

generated more generalizable subgoals performed significantly better at problem solving than any other group (Margulieux, Morrison, et al., 2016). If the students tend to be unable to self-explain, are otherwise struggling in the course (i.e., exhibit signs of already having high cognitive load), or seem unmotivated to learn, then giving subgoal labels would likely be more effective than asking them to generate subgoal labels. Based on whether students generate or are given labels, the transfer distance between worked examples and practice problems can be adjusted to match the most effective conditions.

The instructor's teaching style could also affect how students should engage with subgoal labels. Based on (Margulieux & Catrambone, 2019), students who generated subgoal labels and received feedback on those labels performed better than students who generated subgoal labels without feedback. Therefore, if the instructor's teaching style includes providing feedback or class discussion during which students can refine their generated labels, then generating labels might be more effective than given labels. In contrast, if the course includes a lot of independent learning without many opportunities for feedback or too many students for the instructor to provide individual feedback, then generating labels might be no different than given labels, as was the case in these studies.

The last factor that might determine which type of subgoal learning best suits a course is the curricular materials being used in the course. If the curricular materials, including worked examples and practice problems, were designed by someone else, then the transfer distance between the worked examples and practice problems should determine the type of subgoal learning used. Isomorphic transfer would be best complemented by generating labels, and contextual transfer would be best complemented by given labels (Margulieux, Morrison, & Decker, 2019).

If isomorphic transfer between worked examples and practice problems is an option and the instructor does not have the time or resources to identify subgoal labels for the procedure, then allowing learners to generate subgoal labels for themselves is a good option. To do this, the instructor could use subgoal label training, add a prompt at the end of each problem-solving step, and ask students to generate their own labels. This option would likely be most effective if the instructor matched features between worked examples (e.g., step 2 of the first example is like step 3 of the second example). Margulieux and Catrambone (2019) found that providing hints about which features are similar between worked examples helped students to perform better when they generated their own subgoal labels. Like the feedback described in the paragraph about teaching style, providing hints could further improve the problem-solving skill of learners who are generating their own subgoal labels. It is important to clarify that (Margulieux & Catrambone, 2019) found that providing both hints and feedback did not improve performance; therefore, if feedback or hints are provided, the instructional materials should provide only one or the other.

However, if pre-defined given subgoal labels are used, the worked example – practice problem pairs should utilize contextual transfer to ensure maximum learning. As mentioned earlier, this is contradictory to what would be predicted by cognitive load theory. This is certainly one phenomenon that needs further research. It may be that with given subgoal labels and isomorphic problems students do not adequately self-explain the process associated with each subgoal as the steps are identical within both the worked example and practice problem. To ensure that students in the given subgoal labels with isomorphic practice problems were adequately studying the worked example and attempting the practice problem, we examined the student code submissions. We reviewed student code submissions to ensure that they were not copied from the worked

example. We did a visual inspection and a character by character comparison from the student code submission to the worked example presented. We found no instances of an exact copy of worked example code or any student submissions more than 10% identical to the worked example. Also, the time spent in the instructional period indicates that participants spent similar amounts of time regardless if they received isomorphic or contextual transfer worked example – practice problem pairs.

Conclusion

In this paper we have aggregated the data from three previous studies to take a more holistic view and to examine the results for generalization across populations to provide the most nuanced and accurate information for using subgoal labeled instructional materials in the classroom. By combining the data for maximum statistical power, we can view effect sizes to determine which treatments are likely to yield similar results in the future.

Our research into subgoal labeled instruction in computing represents the first attempt in any discipline (that we are aware of) to test generation of subgoal labels by participants and its effect on learning performance. We are also the first (to our knowledge) to vary transfer between worked examples and practice problems. By introducing these additional conditions into our research, we have found combinations which provide the most beneficial experience for the learner:

1) Given subgoal labeled worked examples with farther transfer between worked examples and practice problems or

2) Student generated subgoal labels for worked examples with nearer transfer between worked examples and practice problems.

Either condition should yield the highest performance from students. Which you choose to implement may depend on conditions discussed above.

Limitations

Our results are limited to having student performance data for only a single lab during an introductory programming course. From this we cannot speculate or generalize to what the long-term impacts are from a learning trajectory perspective. Additionally, the tests were conducted during a single lab session with no delayed test for knowledge over time. Thus, our results only speak to the immediate learning outcomes.

Another potential limitation of this work is the necessary solitary work required of the participants. We asked students in a lab to work alone at their computer for 1-2 hours without assistance from peers or instructors or teaching assistants. This condition was necessary for experimental integrity but is not ecologically valid for many classroom lab environments. While we do not expect that collaboration would negate the learning effects of subgoal labels, it may affect them in unpredictable ways. For example, if some students found similarities between the worked example and practice problem and then helped others in the lab, then the farther transfer intervention might become universally more effective than the nearer transfer intervention. In the study condition where students were asked to generate subgoal labels, if students were working together then the condition would transform from a self-explanation activity to a peer-explanation activity which may or may not benefit each individual student (Chi, 2009).

Future Work

Current research has examined student performance in learning with only a single construct within an entire introductory programming course (`while` loops). Research

has moved from a laboratory environment (Margulieux et al., 2012) to a single lab instance (*Study 1, Study 2, Study 3*) (Morrison, Decker, et al., 2016; Morrison, Margulieux, et al., 2016; Morrison et al., 2015). The next logical step would be to use subgoal labels throughout an entire course and measure student learning. This could be implemented using either of the most beneficial subgoal conditions. Given subgoal labels could be used as long as the worked example and practice problems represented further transfer. Or students could be trained to generate their own subgoal labels and provided with worked example-practice problems with near transfer, and if students receive feedback on their generated subgoal labels to ensure generality.

References

Atkinson, R. K. (2002). Optimizing learning from examples using animated pedagogical agents. *Journal of Educational Psychology*, *94*(2), 416.

Atkinson, R. K., Catrambone, R., & Merrill, M. M. (2003). Aiding Transfer in Statistics: Examining the Use of Conceptually Oriented Equations and Elaborations During Subgoal Learning. *Journal of Educational Psychology*, *95*(4), 762.

Atkinson, R. K., & Derry, S. J. (2000). Computer-based examples designed to encourage optimal example processing: A study examining the impact of sequentially presented, subgoal-oriented worked examples. *Fourth International Conference of the Learning Sciences*. Presented at the ICLS.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, *70*(2), 181–214.

Bransford, J. (2000). *How people learn: Brain, mind, experience, and school*. National Academies Press.

Catrambone, R. (1994). Improving examples to improve transfer to novel problems. *Memory & Cognition*, *22*(5), 606–615.

Catrambone, R. (1995). Aiding subgoal learning: Effects on transfer. *Journal of Educational Psychology*, *87*(1), 5.

Catrambone, R. (1996). Generalizing solution procedures learned from examples. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *22*(4), 1020.

Catrambone, R. (1998). The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General*, *127*(4), 355.

Chi, M. (2009). Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in Cognitive Science*, *1*(1), 73–105.

Chi, M., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, *13*(2), 145–182.

Cohen, J. (1969). *Statistical power analysis for the behavioural sciences.* New York: Academic Press.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Erlbaum: Mahwah.

Eiriksdottir, E., & Catrambone, R. (2011). Procedural instructions, principles, and examples how to structure instructions for procedural tasks to enhance performance, learning, and transfer. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *53*(6), 749–770.

Gravetter, F. J., & Wallnau, L. B. (2016). *Statistics for the behavioral sciences*. Cengage Learning.

Harrington, B., & Cheng, N. (2018). Tracing vs. Writing Code: Beyond the Learning Hierarchy. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 423–428. https://doi.org/10.1145/3159450.3159530

Joentausta, J., & Hellas, A. (2018). Subgoal Labeled Worked Examples in K-3 Education. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 616–621. ACM.

Kalyuga, S. (2011). Cognitive load theory: How many types of load does it really need? *Educational Psychology Review*, *23*(1), 1–19.

Kim, J., Miller, R. C., & Gajos, K. Z. (2013). Learnersourcing subgoal labeling to support learning from how-to videos. *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, 685–690. ACM.

Kumar, A. N. (2015). Solving Code-tracing Problems and Its Effect on Code-writing Skills Pertaining to Program Semantics. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 314–319. https://doi.org/10.1145/2729094.2742587

Margulieux, L. E., & Catrambone, R. (2014). Improving problem solving performance in computer-based learning environments through subgoal labels. *Proceedings of the First ACM Conference on Learning@ Scale Conference*, 149–150. ACM.

Margulieux, L. E., & Catrambone, R. (2019). Finding the best types of guidance for constructing self-explanations of subgoals in programming. *Journal of the Learning Sciences*, *28*(1), 108–151. https://doi.org/doi: 10.1080/10508406.2018.1491852

Margulieux, L. E., Catrambone, R., & Guzdial, M. (2016). Employing subgoals in computer programming education. *Computer Science Education*, *26*, 1–24.

Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012). Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, 71–78. ACM.

Margulieux, L. E., Morrison, B. B., Catrambone, R., & Guzdial, M. (2016). Training Learners to Self -Explain: Designing Instructions and Examples to Improve

Problem Solving. *Transforming Learning, Empowering Learners: The International Conference of the Learning Sciences (ICLS) 2016*, *1*. Singapore.

Margulieux, L. E., Morrison, B. B., & Decker, A. (2019). Design and Pilot Testing of Subgoal Labeled Worked Examples for Five Core Concepts in CS1. *ITICSE'19: Innovation and Technology in Computer Science Education Proceedings*, 7. https://doi.org/10.1145/3304221.3319756

Maxwell, S. E., & Delaney, H. D. (2004). *Designing experiments and analyzing data: A model comparison perspective* (2ne ed.). New York, NY: Psychology Press.

Morrison, B. B., Decker, A., & Margulieux, L. E. (2016). Learning Loops: A Replication Study Illuminates Impact of HS Courses. *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 221–230. https://doi.org/10.1145/2960310.2960330

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). Measuring Cognitive Load in Introductory CS: Adaptation of an Instrument. *Proceedings of the Tenth Annual Conference on International Computing Education Research*, 131–138. https://doi.org/10.1145/2632320.2632348

Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). Subgoals Help Students Solve Parsons Problems. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 42–47. https://doi.org/10.1145/2839509.2844617

Morrison, B. B., Margulieux, L. E., & Guzdial, M. (2015). Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, 21–29. https://doi.org/10.1145/2787622.2787733

Parsons, D., & Haden, P. (2006). Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, 157–163. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

Plass, J. L., Moreno, R., & Brünken, R. (2010). *Cognitive load theory*. Cambridge University Press.

Renkl, A., & Atkinson, R. K. (2002). Learning from examples: Fostering self-explanations in computer-based learning environments. *Interactive Learning Environments*, *10*(2), 105–119.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Silverman, B. (2009). Scratch: Programming for all. *Commun. Acm*, *52*(11), 60–67.

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138.

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive load theory* (Vol. 1). Springer.

Sweller, J., van Merriënboer, J. J., & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296.

Tew, A. E., & Guzdial, M. (2011). The FCS1: A language independent assessment of CS1 knowledge. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 111–116. ACM.

van Gog, T., & Paas, F. (2012). Cognitive Load Measurement. In *Encyclopedia of the Sciences of Learning* (pp. 599–601). Springer.

van Merriënboer, J. J., & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, *17*(2), 147–177.

Wylie, R., & Chi, M. T. (2014). 17 The Self-Explanation Principle in Multimedia Learning. The Cambridge handbook of multimedia learning, 413.

Appendix

In this appendix we supply the majority of our study materials to allow readers to understand exactly what students were given and asked to do.

Demographic Questionnaire

What is your gender? Male Female

What is your age? _____

What is your major? _____

Please report your high school GPA if you remember it: _____ out of _____ (e.g., 4.0)

Which institute are you currently attending? Univ 1 Univ 2 Univ 3

Which best describes how many years you've been in college?

First-year Second-year Third-year Fourth-year Fifth-year
Other: _____

Please report your college GPA if you know it: _____ out of 4.0

Have you taken any computer science courses in high school? Y N

If so, how many ____ and what were their names

_____

Did you take the AP Computer Science test? Y N

If so, what was your score _____

Do you consider English to be your primary language? Y N

How comfortable do you feel solving programming problems?

Not comfortable at all			Neutral			Very comfortable
1	2	3	4	5	6	7

How difficult do you think learning to solve programming problems will be?

Very difficult			Neutral			Very easy
1	2	3	4	5	6	7

Have you solved programming problems using loops before? Yes No

Pre / Post Test

1	Consider the following code segment. value = 15 WHILE value < 28 PRINTLN value value = value + 1 ENDWHILE What are the first and last numbers output by the code segment? first last a. 15 27 b. 15 28 c. 16 27 d. 16 28 e. 16 29 f. I don't know
2	Consider the following code that is intended to print the sum of all values in vals. vals = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12] total = 0 /* missing code */ PRINTLN total Which of the code segments shown below can be used to replace /*missing code */ so that the code will work as intended? I. pos = 0 WHILE pos < length (vals) total = total + vals[pos] ENDWHILE II. pos = length (vals) WHILE pos > 0 total = total + vals[pos] pos = pos - 1 ENDWHILE III. pos = 0; WHILE pos < length (vals) total = total + vals[pos] pos = pos + 1 ENDWHILE a. I only b. II only c. III only d. I and III e. II and III f. I don't know

3	Consider the following code segment:
	
	outer = 0
	WHILE outer < n
	  inner = 0
	  WHILE inner <= outer
	  PRINT outer, “ “
	  inner = inner + 1
	  ENDWHILE
	
	  outer = outer + 1
	ENDWHILE
	
	If n has been declared with the value 4, what is printed as a result of executing the code segment?
	a. 0 1 2 3
	b. 0 0 1 0 1 2
	c. 0 1 2 2 3 3 3
	d. 0 1 1 2 2 2 3 3 3 3
	e. 0 0 1 0 1 2 0 1 2 3
	f. I don’t know

4	Consider the following code segment.
	
	a = 24
	b = 30
	WHILE b != 0
	  r = a % b
	  a = b
	  b = r
	ENDWHILE
	PRINTLN a
	
	What is printed as a result of executing the code segment?
	a. 0
	b. 6
	c. 12
	d. 24
	e. 30
	f. I don’t know

5	Consider the following code segments.
	I. k = 1
	WHILE k < 20
	IF k % 3 == 1 THEN
	PRINT k, “ “
	ENDIF
	k = k + 3

```
        ENDWHILE

II.     k = 1
        WHILE k < 20
            IF k % 3 == 1 THEN
               PRINT  k, " "
            ENDIF
            k = k + 1
        ENDWHILE

III.    k = 1
        WHILE k < 20
            PRINT  k, " "
             k = k + 3
        ENDWHILE
```

Which of the code segments above will produce the following output?

1 4 7 10 13 16 19

a. I only
b. II only
c. I and II only
d. II and III only
e. I, II, and III
f. I don't know

Training

How to Make Subgoal Labels

Research has shown that when studying problem solving procedures, like solving problems using a loop, the best learning happens when students explain to themselves (self-explain) the purpose of steps in the procedure. Successful self-explanations identify the subgoals of the procedure. Subgoals are components of the problem solution (the overall goal) that are made up of individual steps taken to solve the problem (such as adding two numbers together). That means individual steps make up subgoals, and subgoals make up the solution.

For example, if you were asked to solve for x in the equation, $2x + 4 = 6x + 10$, you would use the following steps

Get variables on same side
$2x + 4 = 6x + 10$
$-10 \quad -10$
$-2x \quad -2x$
$4 - 10 = 6x - 2x$

Simplify
-6 = 4x

Get variable with coefficient of 1
-6/4 = 4x/4
-3/2 = x

Each group of steps is a subgoal of the problem. The labels in this example ("Get variables on same side," "Simplify," and "Get variable with coefficient of 1") describe the purpose of the subgoals. A good subgoal label describes the function or the goal of each group of steps. The label should convey what the steps achieve toward solving the problem to help the learner connect steps of the procedure to their purpose.

While you are learning to solve problems using loops, you will be asked to provide your own subgoal labels for the examples that you receive. To do this, you will be asked to identify the purpose of groups of steps in the examples (label the subgoals). Good subgoal labels are action-based phrases (i.e., similarly to imperative sentences like "Close the door," or "Press the button"); they tell the problem solver what to do next. The following activity is intended to give you practice in making your own subgoal labels.

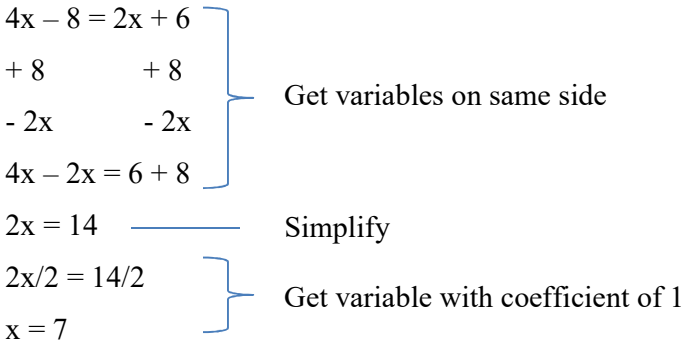
Activity

Group and label the steps of the following example using the same subgoal labels from the previous example.

Solve for x

4x – 8 = 2x + 6
+ 8 + 8
- 2x - 2x
4x – 2x = 6 + 8
2x = 14
2x/2 = 14/2
x = 7

ANSWER
Solve for x



For this order of operations problem, create subgoal labels for each group of steps (by labeling each group of steps with its purpose).

Solve for x

x = 4 * (5-2) + 12 / (4-1) – 7

<Space for creating subgoal>

(5-2) (4-1)

x = 4 * 3 + 12 / 3 – 7

<Space for creating subgoal>

4*3 12/3

x = 12 + 4 – 7

<Space for creating subgoal>

x = 11

ANSWER: Expert created subgoals

Solve for x

x = 4 * (5-2) + 12 / (4-1) – 7

Simplify parentheses (and exponents) operations

(5-2) (4-1)

x = 4 * 3 + 12 / 3 – 7

Simplify multiplication and division operations

4*3 12/3

x = 12 + 4 – 7

Simplify addition and subtraction operations

x = 11

Now that you have some practice applying and creating subgoal labels, it's time to make subgoal labels for solving problems using a loop. The examples that you will be given all have the same subgoals, but this doesn't mean that you have to stick to the subgoal labels that you create on the first example. Please feel free to update your subgoal labels as you learn more about using loops.

Verbal Analogies

Verbal analogies provide excellent training in seeing relationships between concepts. Verbal analogies were previously used to test cognitive ability on standardized tests (like the SAT, the GRE, and other professional exams). Increasingly, too, employers may use these word comparisons on personnel and screening tests to determine an applicant's quickness and verbal acuity.

How to "Read" Analogies

The symbol (:) means "is to" and the symbol (: :) means "as." Thus, the analogy, "aspirin : headache : : nap : fatigue," should be read "aspirin is to headache as nap is to fatigue." Stated another way, the relationship between aspirin and headache is the same as the relationship between nap and fatigue.

Tips for Doing Analogies

- Try to determine the relationship between the complete pair of words.
- Eliminate any pairs in your answer choices that don't have the same relationship.
- Try putting the pairs into the same sentence: "Aspirin relieves a headache." Therefore, a nap relieves fatigue.
- Sometimes paying attention to the words' parts of speech helps. For example "knife" (noun) : "cut" (verb) : : "pen" (also a noun) : "write" (also a verb).

Common Relationships Between Word Pairs

Relationship	Example
Sameness (synonyms)	wealthy : affluent : : indigent : poverty-stricken
Oppositeness (antonyms)	zenith : nadir : : pinnacle : valley
Classification Order (general - specific)	orange : fruit : : beet : vegetable
Difference of Degree	clever : crafty : : modest : prim
Person Related to Tool, Major Trait, or Skill or Interest	entomologist : insects : : philosopher : ideas
Part and Whole	eraser : pencil : : tooth : comb
Steps in a Process	cooking : serving : : word processing : printing

Cause and Effect (or Typical Result)	fire : scorch : : blizzard : freeze
Thing and Its Function	scissors : cut : : pen : write
Qualities or Characteristics	aluminum : lightweight : : thread : fragile
Substance Related to End Product	silk : scarf : : wool : sweater
Implied Relationships	clouds : sun : : hypocrisy : truth
Thing and What It Lacks	atheist : belief : : indigent : money
Symbol and What It Represents	dove : peace : : four-leaf clover : luck

Try a few.

Analogies

1. happiness : smile : : _____ : frown
   - worry
   - terror
   - mood
   - temper
   - encomium

2. water : _____ : : food : hunger
   - element
   - drink
   - starvation
   - liquid
   - thirst

3. government : _____ : : media : news
   - rule
   - bureaus
   - people
   - laws
   - legislature

4. light bulb : electricity : : car : _____
   - oil
   - motor
   - wheels
   - generator
   - gasoline

5. chapter : book : : _____ : nation
   - state
   - country

- kingdom
- president

6. tall : short : : _____ : smooth
   - deep
   - rough
   - texture
   - wide

7. preserve : waste : : ordinary : _____
   - special
   - recycle
   - expensive
   - usual

8. _____ : freeze : : horrible : wonderful
   - stop
   - cold
   - terrible
   - boil

9. _____ : ballerina : : soft : velvet
   - graceful
   - dancer
   - performance
   - edgy

10. genius : _____ : : glass : clear
    - intelligent
    - brains
    - capable
    - slow

Worked Examples

For the first Worked Example we are including the materials for all three conditions: Control, Subgoals Given, and Subgoals Generate. For the remaining problems we are giving only the Control materials.

Worked Example #1 for Sub-Goal CS Study – Calculating Average

Control group (A)

Your friend is working as a server in a restaurant. He has a collection of tips, but wants to know what his average tip is on a Friday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Friday night. What is his average tip?
$15.00, $5.50, $6.75, $10.00, $12.00, $18.50, $11.75, $9.00

Solution

Step one: define and initialize variable to hold the collection of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]        list containing all the tip values
```

Step two: define and initialize variable to hold the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]        list containing all the tip values
sum = 0                                                  accumulator to hold sum of values
```

Step three: initialize the loop to start at the beginning of the list of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0                                                  lcv is loop control variable
# We want to start by looking at the first value in the list which has an index of 0
```

Step four: determine when we are done with the list of tips

```
# In this case we want to process every element in the list, so we will terminate when
we reach the end of the list, or when lcv has the value of length(tips)
lcv >= length(tips)
```

Step five: change the ending condition to be a continuing condition (continue looking at tips while…)

```
# if the termination is when lcv >= length(tips), then to reverse that we have:
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
```

```
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

ENDWHILE
```

Step six: move to the next tip

```
# We want to look at every element in the list so the update of lcv is by one
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

    lcv = lcv + 1
ENDWHILE
```

Step seven: add the current tip to the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]                  as lcv is incremented by one each time
                                          through the loop, the next
   lcv = lcv + 1                          value in the list will be added to sum
ENDWHILE
```

Step eight: calculate the average

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)              the average is the sum of the elements
                                          divided by the number of elements
```

Step nine: print results

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
```

```
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)
PRINTLN average
```

Worked Example #1 for Sub-Goal CS Study – Calculating Average

Subgoal Given group (B)

Your friend is working as a server in a restaurant. He has a collection of tips, but wants to know what his average tip is on a Friday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Friday night. What is his average tip?
$15.00, $5.50, $6.75, $10.00, $12.00, $18.50, $11.75, $9.00

Solution

SUBGOAL: define and initialize variables

Step one: define and initialize variable to hold the collection of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]      list containing all the tip values
```

Step two: define an initialize variable to hold the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]      list containing all the tip values
sum = 0                                               accumulator to hold sum of values
```

SUBGOAL: initialize the loop

Step three: initialize the loop to start at the beginning of the list of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0                                               lcv is loop control variable
# We want to start by looking at the first value in the list which has an index of 0
```

SUBGOAL: determine loop condition

Sub-SUBGOAL: determine termination condition of loop

Step four: determine when we are done with the list of tips

In this case we want to process every element in the list, so we will terminate when we reach the end of the list, or when lcv has the value of *length(tips)*

lcv >= length(tips)

Sub-SUBGOAL: invert the termination condition into a continuation condition

Step five: change the ending condition to be a continuing condition (continue looking at tips while…)

if the termination is when lcv >= length(tips), then to reverse that we have:

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

ENDWHILE
```

SUBGOAL: update loop

Step six: move to the next tip

We want to look at every element in the list so the update of lcv is by one

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

   lcv = lcv + 1
ENDWHILE
```

SUBGOAL: process body of loop (why did we write it?)

Step seven: add the current tip to the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
```

as lcv is incremented by one each time

```
                                    through the loop, the next
   lcv = lcv + 1                    value in the list will be added to sum
ENDWHILE
```

SUBGOAL: determine results

Step eight: calculate the average

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)        the average is the sum of the
                                    elements divided by the number of
                                    elements
```

Step nine: print results

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)
PRINTLN average
```

Worked Example #1 for Sub-Goal CS Study – Calculating Average

Subgoal Generate group (C)

Your friend is working as a server in a restaurant. He has a collection of tips, but wants to know what his average tip is on a Friday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Friday night. What is his average tip?
$15.00, $5.50, $6.75, $10.00, $12.00, $18.50, $11.75, $9.00

Solution

SUBGOAL:

Step one: define and initialize variable to hold the collection of tips

tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9] *list containing all the tip values*

Step two: define an initialize variable to hold the sum

tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9] *list containing all the tip values*

sum = 0 *accumulator to hold sum of values*

SUBGOAL:

Step three: initialize the loop to start at the beginning of the list of tips

tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]

sum = 0

lcv = 0 *lcv is loop control variable*

We want to start by looking at the first value in the list which has an index of 0

SUBGOAL:

Step four: determine when we are done with the list of tips

In this case we want to process every element in the list, so we will terminate when we reach the end of the list, or when lcv has the value of *length(tips)*

lcv >= length(tips)

Step five: change the ending condition to be a continuing condition (continue looking at tips while…)

if the termination is when lcv >= length(tips), then to reverse that we have:

tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]

sum = 0

lcv = 0

WHILE lcv < length(tips) DO

ENDWHILE

SUBGOAL:

Step six: move to the next tip

```
# We want to look at every element in the list so the update of lcv is by one
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

    lcv = lcv + 1
ENDWHILE
```

SUBGOAL:

Step seven: add the current tip to the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
```

as lcv is incremented by one each time through the loop, the next value in the list will be added to sum

SUBGOAL:

Step eight: calculate the average

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)
```

the average is the sum of the elements divided by the number of elements

Step nine: print results

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
```

```
    sum = sum + tips[lcv]
    lcv = lcv + 1
ENDWHILE
average = sum / length(tips)
PRINTLN average
```

Worked Example #2 for Sub-Goal CS Study – Count Matching Values
Control group (A)

You have been told that a pair of dice roll the number 7 more than any other combination. You would like to find out if that's true. To do this you have a pair of dice that you've rolled 20 times. You need to count how many times 7 was rolled.
Here are all the rolls of the dice. How many times was 7 rolled?
2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12

Solution

Step one: define and initialize variable to hold the collection of dice rolls

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]   list containing all the dice rolls
```

Step two: define and initialize variable to hold the count of 7 rolls

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]   list containing all the dice rolls
count = 0                                                                  variable to hold the count of times 7 was rolled
```

Step three: initialize the loop to start at the beginning of the list of dice rolls

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0                                                                    lcv is loop control variable
# We want to start by looking at the first value in the list which has an index of 0
```

Step four: determine when we are done with the list of dice rolls

```
# In this case we want to process every element in the list, so we will terminate when we reach the end of the list,
# or when lcv has the value of length(rolls)
lcv >= length(rolls)
```

Step five: change the ending condition to be a continuing condition (continue looking at dice rolls while…)

```
# if the termination is when lcv >= length(rolls), then to reverse that we have:
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0
WHILE  lcv < length(rolls) DO

ENDWHILE
```

Step six: move to the next roll

```
# We want to look at every element in the list so the update of lcv is by one
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0
WHILE  lcv < length(rolls) DO

    lcv = lcv + 1
ENDWHILE
```

Step seven: determine if the current dice roll is a 7

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0
WHILE  lcv < length(rolls) DO
   IF rolls[lcv] == 7 THEN                          if the current roll is equal to 7

   ENDIF
   lcv = lcv + 1
ENDWHILE
```

Step eight: if the roll is 7, increment the count

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0
WHILE  lcv < length(rolls) DO
   IF rolls[lcv] == 7 THEN                          if the current roll is equal to 7
        count = count + 1                           increment count by one
   ENDIF
   lcv = lcv + 1
ENDWHILE
```

Step nine: print results

```
rolls = [2, 8, 7, 6, 4, 7, 9, 11, 8, 6, 7, 4, 3, 5, 7, 11, 9, 7, 4, 12]
count = 0
lcv = 0
WHILE  lcv < length(rolls) DO
   IF rolls[lcv] == 7 THEN
         count = count + 1
   ENDIF
   lcv = lcv + 1
ENDWHILE
PRINTLN count
```

Worked Example #3 for Sub-Goal CS Study – Counting Prime Numbers

Control group (A)

You are at trivia night at your local favorite eating establishment. One of the questions is "How many prime numbers are between 1 and 100?" No one in the place could answer the question. You decide to write a program to calculate the answer so you can answer it next time! Remember: a prime number is a number that is evenly divisible by only 1 and itself.

Solution

Step one: define and initialize variable to hold the count of prime numbers

```
count = 3                    variable to hold count of prime numbers, 1, 2, and
                             3 are all prime numbers
```

Step two: initialize the loop to start at the first number to test if it's a prime number

```
count = 3
num = 4                      num is the first value we want to test and a loop
                             control variable
# We want to start by looking at the first value to test (we know 1, 2, and 3 are prime
numbers)
```

Step three: determine when we are at the last number to test

```
# In this case we want to test every number up to and including 100
num > 100
```

Step four: change the ending condition to be a continuing condition (continue testing numbers while…)

```
# if the termination is when lcv > 100, then to reverse that we have:
```

```
count = 3
num = 4
WHILE  num <= 100 DO

ENDWHILE
```

Step five: move to the next number to test

```
# We want to test every number up to 100, so we just add one to num
count = 3
num = 4
WHILE  num < 100 DO

    num = num + 1                    adding one to num moves us to the next
                                      number
ENDWHILE
```

Step six: determine if the number is a prime number

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1                     we need to see if the numbers below num go
                                     evenly into num
                                     to do this we need another loop inside our loop
                                     this loop starts at one number less than num
   num = num + 1
ENDWHILE
```

Step sixA: determine if the number is a prime number – initialize variables for inner loop

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1                     we need to see if the numbers below num go
                                     evenly into num
   isPrime = True                    we assume that lcv is a prime number to start with

   num = num + 1
ENDWHILE
```

Step sixB: determine if the number is a prime number – determine continuation condition for inner loop

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO                    we will look at all the values of lcv above 1

   ENDWHILE
   num = num + 1
ENDWHILE
```

Step sixC: determine if the number is a prime number – update the loop control variable

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO

      lcv = lcv – 1                    go down one number for divisor

   ENDWHILE
   num = num + 1
ENDWHILE
```

Step sixD: determine if the number is a prime number – is it evenly divisible?

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO

      IF num % lcv == 0 THEN           if the remainder of num divided by lcv is 0
                                       then it's evenly divisible

      ENDIF
      lcv = lcv – 1
   ENDWHILE
```

```
   num = num + 1
ENDWHILE
```

Step sixE: determine if the number is a prime number – if not evenly divisible, not a prime number

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO

      IF num % lcv == 0 THEN          if the remainder of num divided by lcv is 0
                                      then it's evenly divisible
         isPrime = False              then the number is not a prime number
      ENDIF
      lcv = lcv – 1
   ENDWHILE
   num = num + 1
ENDWHILE
```

Step sixF: determine if the number is a prime number – based on isPrime, add to the count

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO

      IF num % lcv == 0 THEN
         isPrime = False
      ENDIF
      lcv = lcv – 1
   ENDWHILE
   IF isPrime == True THEN            if isPrime is still true, then the number is
                                      really a prime
      count = count + 1               so add one to the count
   ENDIF
   num = num + 1
ENDWHILE
```

Step seven: print results

```
count = 3
num = 4
WHILE  num < 100 DO
   lcv = num - 1
   isPrime = True
   WHILE lcv > 1 DO

      IF num % lcv == 0 THEN
         isPrime = False
      ENDIF
      lcv = lcv – 1
   ENDWHILE
   IF isPrime == True THEN
      count = count + 1
   ENDIF
   num = num + 1
ENDWHILE
PRINTLN count
```

Practice Problems

Example #1 Isomorphic to Solve – Calculating Average

Control group (A)

Your friend is still working as a server in a restaurant. Now he wants to know what his average tip is on a Saturday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Saturday night. What is his average tip?
$20.00, $8.25, $9.75, $6.00, $14.00, $18.50, $10.50, $18.00

Example #1 Isomorphic to Solve – Calculating Average

Subgoal Given group (B)

Your friend is still working as a server in a restaurant. Now he wants to know what his average tip is on a Saturday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Saturday night. What is his average tip?
$20.00, $8.25, $9.75, $6.00, $14.00, $18.50, $10.50, $18.00

Solution
SUBGOAL: define and initialize variables

SUBGOAL: initialize the loop

SUBGOAL: determine loop condition
Sub-SUBGOAL: determine termination condition of loop

Sub-SUBGOAL: invert the termination condition into a continuation condition

SUBGOAL: update loop

SUBGOAL: process body of loop (why did we write it?)

SUBGOAL: determine results

Example #1 Isomorphic to Solve – Calculating Average

Subgoal Generate group (C)

Your friend is still working as a server in a restaurant. Now he wants to know what his average tip is on a Saturday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Saturday night. What is his average tip?
$20.00, $8.25, $9.75, $6.00, $14.00, $18.50, $10.50, $18.00

Solution
SUBGOAL: copied from what they put in WE in order followed by text box for each

SUBGOAL:

SUBGOAL:

SUBGOAL:

SUBGOAL:

SUBGOAL:

Example #1 Context Change to Solve – Calculating Average Control group (A)

As part of a science experiment, you need to determine what the average rainfall was at your house during the past week. You have an accurate rain gage and you've checked it every morning for a week and written down any collection of rain (in inches). Write a program that will calculate the average rainfall.

Here are all the measurements of rain that you collected. What is his average rainfall for the week?
1.5, 0, 0.33, 0.6, 0.95, 0, 0.25

Example #1 Context Change to Solve – Calculating Average Control group (A)

Subgoal Given group (B)

As part of a science experiment, you need to determine what the average rainfall was at your house during the past week. You have an accurate rain gage and you've checked it every morning for a week and written down any collection of rain (in inches). Write a program that will calculate the average rainfall.

Here are all the measurements of rain that you collected. What is his average rainfall for the week?
1.5, 0, 0.33, 0.6, 0.95, 0, 0.25

Solution
SUBGOAL: define and initialize variables

SUBGOAL: initialize the loop

SUBGOAL: determine loop condition
Sub-SUBGOAL: determine termination condition of loop

Sub-SUBGOAL: invert the termination condition into a continuation condition

SUBGOAL: update loop

SUBGOAL: process body of loop (why did we write it?)

SUBGOAL: determine results

Example #1 Context Change to Solve – Calculating Average Control group (A)

Subgoal Generate group (C)

As part of a science experiment, you need to determine what the average rainfall was at your house during the past week. You have an accurate rain gage and you've checked it every morning for a week and written down any collection of rain (in inches). Write a program that will calculate the average rainfall.

Here are all the measurements of rain that you collected. What is his average rainfall for the week?

1.5, 0, 0.33, 0.6, 0.95, 0, 0.25

Solution
SUBGOAL: labels from they put in WE1 followed by text area

SUBGOAL:

SUBGOAL:

SUBGOAL:

SUBGOAL:

SUBGOAL:

Example #2 Isomorphic – Count Matching Values

Control group (A)

Now you want to know how many times someone rolls snake eyes (two 1's) on a pair of dice. Write the program that will count how many times 2 is rolled out of 20 rolls. Here are all the rolls of the dice. How many times was 2 rolled?

2, 8, 7, 6, 2, 7, 9, 11, 8, 6, 7, 2, 3, 5, 7, 11, 2, 7, 4, 12

Example #2 Context Change – Count Matching Values

Control group (A)

A friend is trying to figure out where to go for lunch. She has a bike, but doesn't want to bike more than 3 miles away because it might take too long. She has asked you how many lunch places there are within 3 miles. Another friend has already looked up the distances to all the food places listed on Yelp! and you need to find out how many are less than 3 miles away.

Here are all the distances to the nearby restaurants. How many are less than 3 miles away?
5.2, 3.6, 2.4, 1.0, 2.2, 8.9, 6.0, 2.9, 4.3

Example #3 Isomorphic – Counting Prime Numbers

Control group (A)

Oh those wily characters who run trivia night – they changed the question! This time they asked how many prime numbers are between 100 and 200 (inclusive). But now you write a program to answer immediately! Remember: a prime number is a number that is evenly divisible by only 1 and itself.

Example #3 Context Change – Counting Unique Values

Control group (A)

So you just bought a new cell phone and the nice salesperson at the store imported all of your contacts into the new phone. However there must have been an error in the algorithm that they used because now you have duplicates of some of your contacts! You need to know how many contacts you have because you need to order invitations to your graduation. Can you find out how many unique values are in your contacts? You can assume that all your contacts have a unique integer identifier.

Here is the list of unique integer identifiers. Write the program to count the number of identifiers that only appear once in the list.

1234, 2356, 6347, 1264, 3678, 1234, 6378, 2356, 1637, 2734, 1264, 9654, 8888, 9654, 2346, 1264

Assessment Problems

Assessment #1 – Calculating Average

You are a Teaching Assistant for a class. The instructor has given you a collection of quiz grades and asked you to record the grades and calculate the class average. Write a program that will calculate the average quiz grade for the class.

Here are all the quiz grades for the class. What is the average score?
90, 80, 75, 60, 95, 92, 88, 41, 50, 85, 90, 85

Assessment #2 – Average Matching Values

You are a Teaching Assistant for a class. The instructor has given you a collection of test grades and asked you to record the grades and calculate the class average *for passing*

grades (those that are 70 or above). Write a program that will calculate the average passing test grade for the class.

Here are all the test grades for the class. What is the average passing score?
90, 80, 75, 60, 95, 92, 88, 41, 50, 85, 90, 85

Assessment #3 – Sum Golf Scores

Your best friend is a big-time golfer, but is not very good at math. He continues to make errors when adding up his scores. You volunteer to write a program that will add up his golf scores and print out the scores for the first nine holes, the second nine holes, and total for the round.

Here are your friend's totals for one round of golf:
4, 3, 5, 6, 3, 4, 7, 5, 4, 3, 4, 4, 5, 6, 7, 4, 5, 6

Assessment #4 – Population

Suppose that a certain group's population grows at a rate of 10% every year. Write a program that will determine how many years it will take for the population to double. Suppose that currently there are 100 specimens in the current population.

Parsons Problem

Here is another question along with the steps to the solution. However the steps are not in the correct order. Please put the steps into the correct order by numbering them (put a 1 by the steps that go first, a 2 by the steps that go in the second, etc.).

Rainfall Problem
Let's imagine that you have a list that contains amounts of rainfall for each day, collected by a meteorologist. Her rain gathering equipment occasionally makes a mistake and reports a negative amount for that day. We have to ignore those. We need to write a program to:

a) calculate the total rainfall by adding up all the positive integers (and only the positive integers),
b) count the number of positive integers, and
c) print out the average rainfall at the end. Only print the average if there was some rainfall, otherwise print "No rain".

_____ lcv = lcv + 1

_____ IF count > 0 THEN

_____ ENDIF

_____ WHILE lcv < length(rain) DO

_____ ELSE

_____ sumRain = sumRain + rain[lcv]
count = count + 1

_____ lcv = 0

_____ ave = sumRain / count
PRINTLN ave

_____ rain = [0,5,1,0,-1,6,7,-2,0]
sumRain = 0
count = 0

_____ PRINTLN "No rain"

_____ IF rain[lcv] >= 0 THEN

_____ ENDIF

_____ ENDWHILE

	No Subgoal Labels		Subgoal Labels Given		Subgoal Labels Generated	
	Consent					
	Demographics*					
	Pre test*					
	Training Problem – summing					
	Analogy Training & Activity*				Subgoal Training & Activity*	
Groups	**No Subgoal Isomorphic Transfer**	**No Subgoal Contextual Transfer**	**Given Isomorphic Transfer**	**Given Contextual Transfer**	**Generate Isomorphic Transfer**	**Generate Contextual Transfer**
Worked Example 1 - Calculate Average Tip	(no subgoal labels)*		(subgoal labels given)*		(space to generate subgoal labels)*	
Problem 1 - Calculate Average	P1 - Tip* (no subgoals)	P1A - Rainfall* (no subgoals)	P1 - Tip* (given)	P1A - Rainfall* (given)	P1 - Tip* (generate)	P1A - Rainfall* (generate)
Worked Example 2 - Count Matching Values (dice - 7)	(no subgoal labels)*		(subgoal labels given)		(space to generate subgoal labels)	
Problem 2 - Count Matching Values	P2 - Dice-2* (no subgoals)	P2A - < 3 (no subgoals)*	P2 - Dice-2 (given)	P2A - < 3 (given)	P2 - Dice - 2 (generate)	P2A - < 3 (generate)
Worked Example 3 - Count Prime Numbers (1-100)	(no subgoal labels)*		(subgoal labels given)		(space to generate subgoal labels)	
Problem 3 - Count	P3 - Primes (100-200) (no subgoals)	P3A - Unique Values (no subgoals)	P3 - Primes (100-200) (given)	P3A - Unique Values (given)	P3 - Primes (100-200) (generate)	P3A - Unique Values (generate)
	Cognitive Load Measurement					
	Problem Solving Assessment (4 problems; 2 near transfer, 2 far transfer)*					
	Parsons Problem Assessment *					
	Post Test *					

Figure 2. Complete study procedure. Items with * are provided in the Appendix.

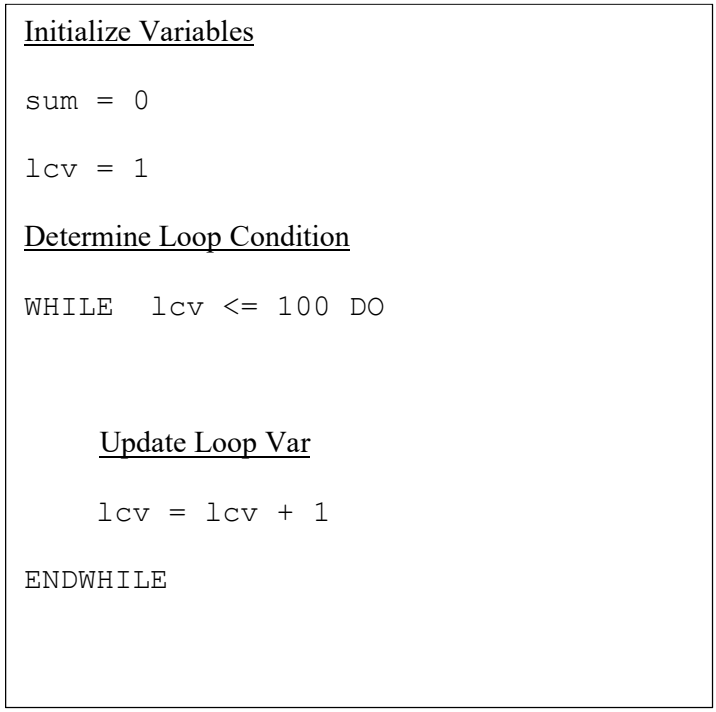


Figure 1. Partial worked example illustrating subgoal labels. Subgoal labels are underlined.

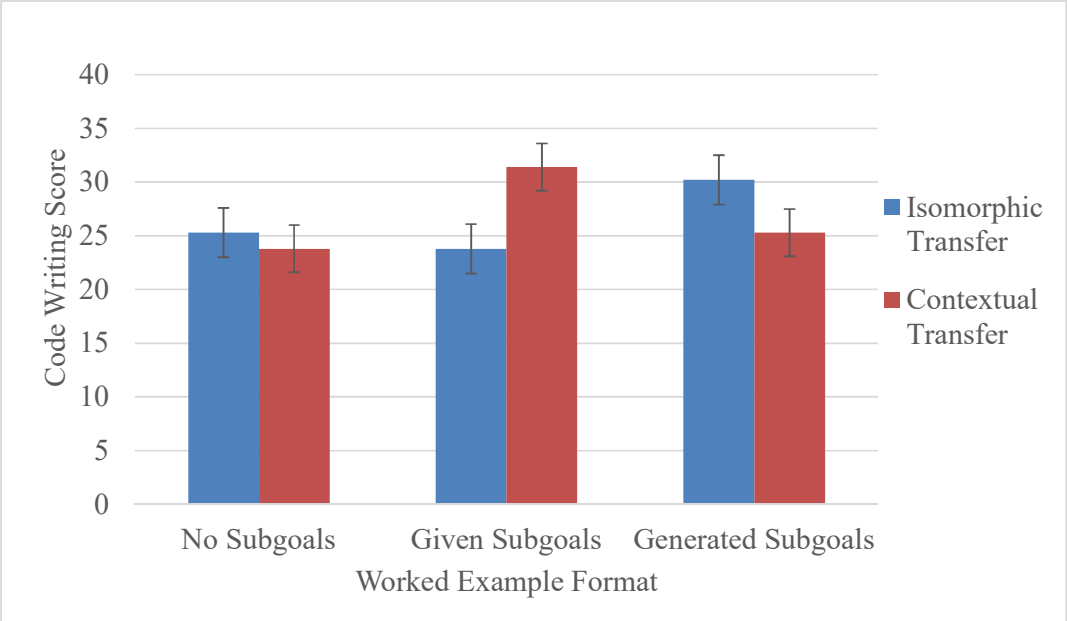


Figure 3. Performance across six groups on problem-solving (code writing) tasks.

Maximum score was 44. Error bars represent standard error.

Table 1. Learner Characteristics in Experiment 1 and Their Relationship to Performance.

Learner Characteristic	Mean/Proportion	Std. Deviation	Correlation with Problem-Solving Performance
Gender	84% male	-	$\rho = -.02$, $p = .90$
Age	21	4	$r = -.06.$, $p = .60$
Major	50% CS major	-	$\rho = -.03$, $p = .78$
High School GPA	3.40	0.58	$r = -.06$, $p = .59$
College GPA	3.08	0.55	$r = .18$, $p = .13$
English is primary language	91% yes	-	$\rho = .06$, $p = .61$
Years in college	2.4	1.4	$r = -.03$, $p = .81$
Comfort with computers*	4.2	1.5	$r = .46$*, $p < .001$
Expected difficulty of task*	4.0	1.4	$r = .29$*, $p = .007$
Prior course in programming	42% yes	-	$\rho = .37$*, $p < .001$

*Note: The question about comfort with computers asked student to rate how comfortable they were using a computer on a 7-point scale that ranged from “1 - not comfortable at all” to “7 - very comfortable.” The question about expected difficulty of task used a 7-point scale that ranged from “1 - very difficult” to “7 - very easy”.

Table 2. Learner Characteristics in Experiment 2 and Their Relationship to Performance.

Learner Characteristics	Mean/Proportion	Std. Deviation	Correlation with Problem-Solving Performance
Gender	40% male	-	$\rho = .05$, $p = .79$
Age	22	6.9	$r = .05.$, $p = .79$
Major	23% CS major	-	$\rho = .01$, $p = .94$
High School GPA	3.81	0.37	$r = .43$*, $p = .02$
College GPA	3.33	0.53	$r = .15$, $p = .42$
English is primary language	91% yes	-	$\rho = .32$, $p = .06$
Years in college	3.3	1.8	$r = -.14$, $p = .42$
Comfort with computers*	3.5	1.2	$r = .07$, $p = .68$
Expected difficulty of task*	3.2	1.3	$r = .24$, $p = .17$
Prior course in programming	29% yes	-	$\rho = .11$, $p = .52$

Table 3. Learner Characteristics in Experiment 3 and Their Relationship to Performance.

Learner Characteristics	Mean/Proportion	Std. Deviation	Correlation with Problem-Solving Performance
Gender	71% male	-	ρ = -.02, p = .90
Age	19	3	r = .08., p = .46
Major	33% New Media 63% Game Design	-	ρ = .11, p = .30
High School GPA	3.61	0.32	r = .05, p = .70
College GPA	3.47	0.62	r = -.06, p = .64
English is primary language	96% yes	-	ρ = -.15, p = .15
Years in college	1.9	0.8	r = .06, p = .57
Comfort with computers*	5.3	1.4	r = .52*, p < .001
Expected difficulty of task*	4.5	1.4	r = .31*, p = .002
Prior course in programming	94% yes	-	ρ = .30*, p = .003

Table 4. Pairwise comparisons evaluating simple main effect of worked example format.

Transfer Distance	Worked Example Format Comparison	Std. Error	Mean Difference	Significance	Effect size (*d*)
Isomorphic	*No Subgoal* to *Given*	3.34	1.55	.64	.10
	No Subgoal to *Generate*	3.34	-4.80	.15	.37
	Given to *Generate*	3.52	-6.36	.02	.44
Context	*No Subgoal* to *Given*	3.01	-7.62	.01	.59
	No Subgoal to *Generate*	3.15	-1.49	.64	.11
	Given to *Generate*	3.24	6.14	.06	.47

Table 5. Correlation between Time on Task and Performance within Each Experimental Group.

Experimental Group	No Subgoal	Given Subgoal	Generate Subgoal
Isomorphic Transfer	$r = 0.26$, $p = .140$	$r = 0.67$, $p < .001$	$r = 0.51$, $p = .008$
Contextual Transfer	$r = 0.53$, $p = .001$	$r = 0.41$, $p = .011$	$r = 0.27$, $p = .186$