A Pedagogically-Focused Translation Pipeline for Designing Loops in Programming

L. Joshua Crotts
Department of Computer Science
Indiana University, Bloomington
Bloomington, USA
ljcrotts@iu.edu

Chung-chieh Shan
Department of Computer Science
Indiana University, Bloomington
Bloomington, USA
ccshan@iu.edu

Sam Tobin-Hochstadt
Department of Computer Science
Indiana University, Bloomington
Bloomington, USA
samth@iu.edu

***Abstract*—This full innovative practice paper describes a step-by-step translation process for loop construction that beginning programming students can follow. Loops in programming languages are difficult for beginning students to correctly construct. At their core, the idea is simple: repeat a given task until a condition is falsified. Loops, however, have multiple possible points of failure and opportunities for mistakes. Via a series of correctness-preserving transformations, students engage with an fully-mechanical translation pipeline that tries to removes many instances of failure and frustration when designing loops. We perform two evaluative studies on students from the CS1 and CS2 population at a large R1 midwestern university to test our methodology against the traditional approach to teaching loops in a second-semester computer science course. The first study emphasized translating tail recursive functions into loops. The second study, which builds upon the results from the first, moves students from a problem statement, a recursive function, then to its loop counterpart. The results from our first study indicate that our translation pipeline method improves students' success in translating tail recursive functions to equivalent loops. The results of our second study indicate that more students appear to enjoy the presentation of the translation pipeline versus others' attitudes toward the traditional means of learning loops. Combining these facts together can potentially imply that students who use the translation pipeline perform better at writing loops than those trained by traditional lectures.**

***Index Terms*—Computer science; Undergraduate; Instructional methods**

I. Introduction

Introductory computer science and programming courses often teach loops to students. Unfortunately, this introduction often runs into confusion and difficulty for several reasons:

- **Problem specification.** Students frequently run into trouble when given a programming exercise to solve by virtue of not knowing how to integrate loops into a correct solution. A problem specification is often intentionally broad to impose critical thinking, but this may result in unintended consequences by overwhelming a student with uncertain design choices.
- **Unintuitive syntax.** Programming languages couple loops with specific keywords and structure that students may misuse and conflate with other meanings.
- **Conditional confusion.** A core component of loop design is determining its ending condition, which may come with students improperly negating conditions and misunderstanding what it means for a loop to terminate. This also relies on students fully and correctly understanding how to negate boolean expressions.
- **Variable scoping.** Local variables typically govern the state of a loop, which consequently may have students shadowing (hiding) variables or attempting to use undeclared variables.
- **Side-effects and variable mutation.** The result of a loop condition changes via variable updates (e.g., `i++`) or some other side-effect-inducing code (e.g., setting a flag to be true or closing a file resource). This causes students to have to micromanage multiple program states.
- **Shotgun debugging.** Beginning students rarely debug with the proper tools to do so, and instead resort to flipping arbitrary conditions and assigning variables at random with the hopes of fixing bugs.

In common with the reasons above, the cognitive load induced by such a staggering jump in complexity may be too intense for some students to handle. Literature [1] suggests that a step-by-step, or scaffolded, pedagogical approach may be an improved alternative to the current method of directly teaching loops from a blank slate. In this paper, we propose such a scaffolded approach via a translation pipeline that starts students off with a problem that they design a recursive solution for, which leads to a tail recursive algorithm, and finally a loop. The idea is that, by scaffolding students through a series of correctness-preserving transformations [2], they are able to write coherent and correct loops that avoid the common pitfalls as outlined above. To this end, our contributions encompass and answer two research questions:

RQ1: Does our translation pipeline method help students that already understand recursion and conditionals to learn to write loops?

RQ2: Why is our proposed method to teach students that already understand recursion and conditionals an improvement over the traditional pedagogy?

II. Prior Work

Our research connects to prior work across many areas of introductory programming education. In this section we discuss the relation to four areas of prior work: (1) those that teach/emphasize functional programming and recursion

prior to loops, (2) those that present tools or pedagogical technique to teach loops directly, (3) those that use of subgoals or scaffolded knowledge when teaching loops, and (4) those that discuss the transfer of knowledge across iteration and recursion.

A. Teaching recursion before loops

Numerous curricula propose teaching functional programming, without traditional loops at all, in a variety of settings, ranging from middle school [3] to secondary school [4] to CS1 in college [5], [6]. These proposals typically do not discuss the transition process to programming with loops in a different language. Some have proposed curricula for a second course that uses loops, but without explicit discussion of how to write loops or evaluation of success [7], [8].

Santos et al. wrote an experience report on the transition from a course in Racket to Java identical to the Northeastern University curriculum [9]. Their report explains four broad opportunities for misconceptions and mistakes: structures to classes, `cond` to `if`, unchecked documentation comments to compiler-checked types, and fixity plus overloaded operators with type coercion. Surprisingly, the report makes no mention of loops or any recursion/loop correspondence.

As noted by Turbak et al., historically, computer science courses and textbooks that teach Java most often place the topic of recursion closer to the middle or end [10]. They argue for teaching recursion prior to loops by presenting the *divide-conquer-glue* paradigm of constructing a solution. To this end, they discuss and emphasize that tail recursion introduces no new mandatory syntax, both of which are ideas that we reiterate in our paper. They conclude with a call for computer science instructors to teach recursion before loops.

In Joosten et al., they describe a curriculum and initiative to teach functional programming before imperative programming, as had been historically so at the University of Twente [11]. This curriculum entails three terms over one year, divided into functional programming, then imperative, then a mix. The researchers then present a study where students are divided into either a 'functional' (experimental) or 'imperative' (control) group for the course. After a pre-test and post-test, their data show that the 'functional' group developed more functions and crafted a stronger mental correspondence with the design and implementation than those of the 'imperative' group.

Lu presents the results of two studies, one switching between two functional languages without imperative loop constructs (Racket and Pyret) and one from a functional language to an imperative one with explicit loops (Pyret and Python) [12]. These studies focus on how well knowledge transferred between languages, but provide little information about loops.

B. Pedagogical support for writing loops

Given the centrality of loops in many programming languages and problems, researchers have proposed a variety of approaches to aid students in constructing loops.

Makri et al. created the "*Loop*" game, which is a two-dimensional puzzle game developed in Unity where the objective is to control a mermaid as they collect coins in the environment [13]. Players control the mermaid through conditionals, loops, and control flow keywords. The researchers surveyed students in a software engineering course and assessed them through a pre-test and post-test, occurring before and after playing the game. Their results indicate that the game is a helpful supplementary tool for learning loops. Tr et al. similarly present the "Ball Targeting Game" which requires students to use program construction, including while loops, to play the game [14]. The game is used in a college-level programming course, but specific results are not presented.

Gomes et al. describes a visual metaphor for presenting loops to novice students and those who struggle in an introductory programming course [15]. A study of code comprehension and ability to modify existing code when using the visual representation showed slight improvement for these populations. Additionally, students expressed satisfaction with these presentations. Cetin et al. describes a visualization approach used while teaching pre-service teachers to construct loops. This resulted in increased success with loop construction when the visualization was used during the teaching process [16]. None of the experiments discussed provided students with explicit assistance in constructing loops, however.

Finally, and most importantly as a link to this study, Morazán demonstrates a similar translation mechanism of translating recursive functions into 'state-based functions' with while loops [17] in the Racket programming language. Their analysis includes several mentions to invariants and a call back to the "design recipe" for functions by Felleisen et al. [6]. Their analysis contains no mention of any formal testing and experimentation. As part of their future work, they also hope to incorporate advanced topics like continuation-passing style. Their explicit use of invariants and control-flow structures like continuations, paired with the narrowed scope of only using Racket, limits the generalizability of the approach and appeal to students learning loops for the very first time (as presenting students with such an advanced technique is, at best, intimidating).

C. Scaffolding

In Morrison et al., they propose a sub-goal labeling methodology [18]. A sub-goal, as defined by [19], is a pedagogical technique to break down a problem into sub-components. Morrison's work places either expert-written labels or placeholders for students to fill in at certain points of a loop definition. The scaffolding, namely the subgoals, is to aid in the problem-solving process.

D. Transfer of knowledge

Kessler et al. designed an experiment evaluating the transfer of knowledge of students that were trained on recursion/iteration, then tested on their ability to write code, with randomly-assigned conditions (i.e., some students were trained on

recursion and had to write an iterative function) [20]. Their data concluded that students who were trained with (were shown an example of) recursion and then were asked to write an iterative function struggled much more than those of the reverse. They also speculate that "...it is good to teach iterative programming before recursive programming for students who have had no prior experience..." [20, p. 164], a claim that has since been relentlessly challenged by other experience reports.

Wiedenbeck et al. performed a similar example-driven experiment to that of Kessler et al., in that students in a freshmen-level computer science course were presented with recursive or iterative functions that are isomorphic in theory but not in name to mathematical constructs, e.g., factorial [20], [21]. They mention that their work aims to present iteration and recursion from a non-programming context. These students were then asked to trace out invocations of these functions. To determine a transfer of knowledge, students were subsequently assigned randomly to trace out either more recursive/iterative invocations, or were swapped to the opposite group. Their data is indicative that students learn best through examples, and the aid of either prior knowledge in recursion or iteration does not provide extensive help in learning the other.

III. Our Approach to Teaching Designing Loops

We propose to teach students to design loops in 3 phases. We first illustrate the steps with an example, then discuss details in greater generality. To our knowledge, this is the first time such a technique has been used in the pedagogical literature with evidence of its efficacy.

A. Translation Example

Although our method works to produce loops over any arbitrary data structure, for simplicity we choose to first illustrate it with a "list product" function. Suppose the student receives the following problem to solve: "Design a function that takes a list of integers and computes its product. The product of the empty list is 1."

(1) Without worrying yet about using a loop, in phase (1), the student designs a recursive function that solves the problem. This solution is shown in Listing 1. Coming up with this solution is straightforward, because a list can be sliced into its head and tail, and the tail (i.e., the rest of the list) is typically the correct argument to pass to the recursive call [6, §9.3]. Another possible solution is to use indices and structurally recurse over the index-of-interest. This is not only more efficient but also easier to apply to other languages without inherent slice syntax.

Listing 1: Recursive List Product Def.

```
def prod(ls):
  if ls == []:
    return 1
  else:
    return ls[0] * prod(ls[1:])
```

(2) In phase (2), The student checks if the function just designed is *tail recursive*. It is not, so the student designs a tail recursive function to solve the same problem. This solution is shown in Listing 2. It uses an accumulator [6, §32.2] called `acc`. This design process includes writing a comment (the first line of Listing 2) that explains the value of each accumulator. This comment is called the *accumulator statement*. In Python, we can use default arguments to supply a default accumulator value. Languages without default arguments need a *driver function* to jump-start the tail recursive *helper function*.

(3) In phase (3), to kick off the translation to a loop, the student marks three pieces of information in the tail recursive function: (i) the base case condition(s), (ii) the non-recursive return value(s), and (iii) each updated parameter in a recursive call.

Listing 2: Highlighting Pieces of List Product Def.

```
# acc: keeps track of the current value of product.
def prodTR(n, acc = 1):
  if ls == []:
    return acc
  else:
    return prodTR(ls[1:], acc * ls[0])
```

(3.1) Now the student is ready to write the loop, progressively in the remaining 4 steps. In the current step, the student converts the accumulator `acc` into a local variable declared at the top of the loop definition. Above the local variable declaration, the student places the accumulator statement, now called an *iterative variable purpose statement*. This comment helps to explain why, for list product, the initial `acc` should be `1`.

Listing 3: Localizing Accumulators for List Product Def.

```
def prodLoop(n):
  # acc: keeps track of the current value of the product.
  acc = 1
```

(3.2) The student writes the `while` keyword, followed by the logical negation of our base case conditions via the `not` operator (or the respective negation operator).

Listing 4: Localizing Accumulators for List Product Def.

```
def prodLoop(n):
  # acc: keeps track of the current value of the product.
  acc = 1
  while not (ls == []):
```

(3.3) The student turns updated parameters into assignment statements within the loop body. The order of assignment statements matters, as we discuss below.

Listing 5: Updating Variables for List Product Def.

```
def prodLoop(n):
  # acc: keeps track of the current value of product.
  acc = 1
  while not (ls == []):
    acc = acc * ls[0]
    ls = ls[1:]
```

(3.4) Lastly, the student adds a `return` statement to return each non-recursive return value of the tail recursive function.

Listing 6: Returning Atomic Value for List Product Def.

```
def prodLoop(n):
  # acc: keeps track of the current value of product.
  acc = 1
  while not (ls == []):
    acc = acc * ls[0]
    ls = ls[1:]
  return acc
```

This completes the example of turning the list product problem statement into a recursive solution, then tail recursive, and finally into a loop. The next section generalizes the translation to work on (almost) any recursive function.

1) General Translation Pipeline: Figure 1 depicts our translation pipeline in general. The steps of the pipeline serve as scaffolding to reduce the cognitive load of loop design until the student eventually becomes able to write a loop directly.

1) **Recursive solution.** The first step is for the student to solve the problem recursively. Teaching this to beginning students can be easier than throwing them into the deep end of writing a loop directly, even in a language with control constructs that make writing loops convenient for seasoned programmers. This is because the design of a recursive function can itself be scaffolded, and often follows the structure of the input data naturally [6].
2) **Tail recursive solution.** The design of a tail recursive solution can also be scaffolded. To develop the solution, the student usually needs to introduce one or more *accumulators*, which are additional arguments to the function that hold context or history, i.e., state. For each accumulator introduced, the scaffolding requires the student to write a comment describing what purpose it serves and what value it stores.[1]
   Whereas the design of the recursive and tail recursive solutions has been addressed in the literature, the remaining steps are novel.
3) **Highlighted pieces.** Three crucial pieces of the tail recursive solution will comprise the iterative solution and should be first identified by the student. The key to identifying the pieces is to classify each return statement as either recursive or not. For a non-recursive return, the student highlights the condition for reaching it (i.e., the termination condition) and the value it returns. For a recursive return, the student highlights each argument to the (tail-)recursive call that is updated rather than merely copied verbatim (e.g., `return prodTR(ls, acc)`).
4) **Iterative variable.** To start off the iterative solution, the student declares a local variable for each accumulator argument to the tail recursive function. The student documents and initializes the variable in the same way the accumulator was documented and initialized.
5) **Loop condition.** To open the loop, the student writes "`while !(`C`):`", where the negated C is the base case condition (or their disjunction, if multiple exist).
6) **Loop body.** Finally the student reaches the body of the loop, which will consist of assignment statements that originate from updated parameters. To decide what variables to update and in what order, the student should abide by three rules respectively:
   - **Rule of reassignment**: If a tail recursive call passes an expression e to update a parameter p, then in the loop body, simply put the local assignment statement $p = e$.
   - **Rule of update**: If a tail recursive call updates two parameters p and q, and p's value is used to update q, then the loop body must assign to q before assigning to p.
   - **Rule of temporary**: If a tail recursive call updates two parameters p and q, where the two variables depend on each other, then we must introduce a temporary variable t_p, assign p to it, update p, then use t_p when updating q.

   The rule of update ensures that a variable that other updates depend on is not prematurely updated. To understand and apply this rule, the student can trace variable values along loop iterations. For instance, in the list product loop, the two possible ways to order the updates to `ls` and `acc` can be traced as in Figure 2: The variable update order on the left violates the rule of update, because `ls` is used to update `acc`. Consequently, the trace on the left does not match how the tail recursive solution runs, and the result 0 does not match what the tail recursive solution returns. Seeing this mismatch should prompt the student to try the opposite order, on the right. This order abides by the rule of update, because `acc` is assigned before `ls` is. Accordingly, it matches the tail recursive solution.

   It may take the student a few tries to determine the correct order of update according to our rules. The goal is to match a tail recursive trace done previously that is known to be correct.
7) **Return.** At the end of the loop, the student adds return statement(s) that return the accumulated result.[2]

IV. Methods

In this section we provide a description of two research studies performed to test the efficacy of the translation pipeline.

1) Research Study #1: Our first research study was motivated by two goals, both of which correspond to our research

[1] *Any* recursive function can be mechanically transformed into a tail recursive solution [22], but we do not expect the student to do this step mechanically. In the list product example, relating the accumulator `acc` to the mechanical transformation requires multiplicative associativity.

[2] We omit one part of this step: if there are multiple base cases, then there should be a case analysis after the loop that denotes what base case value to return.

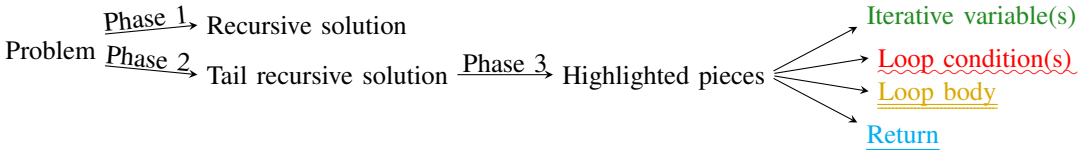

Fig. 1: General Translation Pipeline

```
ls = ls[1:]
acc = acc * ls[0]
```

It #	ls	acc
0	[2, 8, 3]	1
1	[8, 3]	1
2	[3]	8
3	[]	24

```
acc = acc * ls[0]
ls = ls[1:]
```

It #	ls	acc
0	[2, 8, 3]	1
1	[8, 3]	2
2	[3]	16
3	[]	48

Fig. 2: Violating and abiding by the rule of update

questions. Firstly, we aimed to examine whether there was an upward trend of growth in students' abilities to write loops after having viewed our translation pipeline materials. Secondly, we wanted to evaluate our pedagogical technique against standardized lecture materials that students generally encounter when first learning loops.

a) Study Description & Activities: We recruited students from the undergraduate and graduate populations of a computer science/informatics/engineering program at a large R1 midwestern university. To ensure that participants had exactly the prerequisite knowledge (i.e., variables, conditionals, functions, simple data structures, and recursion), we imposed two requirements: participants must be eighteen years of age or older, having taken one or, at most, two computer science courses. We distributed an email to the computer science listserv and advertised the study across two CS1 and CS2 courses. Participants that completed all components of the study received a $25 Amazon gift card.

We randomly assigned each participant to either the experimental or control group. The groups differed only in what lecture materials they had to read. In particular, the experimental group received print-out slides of designing loops through an older version of the pipeline. The control group read through a set of slides about loops from an introductory Java textbook. To evaluate whether the materials caused a significant difference, we tested participants both before and after the intervention. In each test, participants had to convert two (tail) recursive functions into ones that use a loop of some kind. Aside from disallowing recursion, we imposed no other problem-solving restrictions. Participants were not required to completely answer a question before moving on.

The pre-test gauged their ability to work with conditionals and recursion as a baseline comparison. Participants took anywhere from ten to thirty minutes to finish the pre-test. Following the pre-test, we provided to them either the experimental or control materials. Participants took, on average, thirty minutes to read the material, with the control group taking longer due to a higher number of slides. Finally, the participants took a post-test that was similar to the pre-test, just with different tail-recursive functions. (See Section A to view the questions used in both tests).

b) Results: The tests were scored out of 1 point each, with point values awarded for correctness of the loop condition, variable updates, return values, and parameters. Figure 3 plots the scores of the experimental group ($n = 12$) and the control group ($n = 10$). The x-axis shows pre-test scores and the y-axis shows post-test scores. Both groups improved: every post-test score is greater than or equal to the pre-test score. The experimental group improved more: all but two participants scored perfectly on the post-test, whereas in the control group, five of the ten participants performed only marginally better than their pre-test scores.

As a qualitative analysis, we examined the most common mistakes made by participants in both groups. More participants in the experimental group correctly negated the loop condition than those in the control. In particular, when a `while` loop was used, students often incorrectly flipped the relational operator from < to >, instead of >=. Additionally, a few participants changed the lower bound on the number of iterations of a loop from 0 to -1 to align with the choice of >. While this is a correct modification in our test scenarios, we envision that students may encounter bugs by making such decisions. In contrast, every student in the experimental group wrote a correct terminating condition, whether this was via an infinite loop (with a `break`), a bounded `for` loop, or a correct negation (either via `not` or correct DeMorgan's law application).

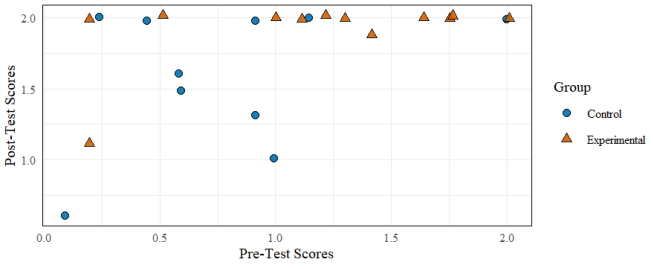

Fig. 3: Experiment 1: Pre-test vs Post-test Scores by Group

As a quantitative/statistical analysis as performed in a similar experiment by [4], we performed a two-sided Mann-Whitney U test for statistical significance on the difference

in pre-test and post-test scores. (The Mann-Whitney U test does not require the samples to be normally distributed, but the data must come from independent samples.) We subtracted each pre-test score from each post-test score, which produced a statistically-insignificant result ($W = 66$, $p = 0.716$). We believe this to be due to the low sample size.

A Mann-Whitney U test was performed to evaluate whether the post-test scores differed by group. The results indicated that there was a weakly-significant difference, $W = 38$, $p = 0.087$.

To ensure that our samples start from the same distribution, we ran a Mann Whitney U test on the pre-test data. The results indicated that there was no significant difference between the pre-test scores of the experimental and control group, $W = 36$, $p = 0.121$.

2) Research Study #2: To gain a deeper understanding about the effects of the translation pipeline on student learning, we conducted an expanded-upon second empirical evaluation. We wanted to see how students learn to design loops from the beginning of the pipeline to the end. That is, when given a problem statement, are they capable of correctly designing a recursive function, then an equivalent loop.

a) Study Description & Activities: Like the first study, we recruited participants from the undergraduate population of a large R1 midwestern university from the computer science, informatics, and engineering school. Participants had to be familiar with CS1 topics, including recursion, to take part. Our prescreening survey included two pseudocode problems where potential participants must trace recursive code. Only those who got at least once answer correct were allowed to participate. We also filtered out those participants with "too much experience," as described in experiment #1. Participants that fully completed the study received a $25 Amazon gift card.

Also like the first study, we used a pre-test, note-reading, and post-test modality. The pre and post-tests each had two Python-based programming questions, which were randomly chosen from a selection of recursive and loop-based programming problems from several popular computer programming textbooks. For each problem, the participants had to write both a recursive solution and a loop solution, in either order.

Participants were randomly assigned to one of two groups. The only difference between the two groups was what material they read. The experimental group read a refined version of the translation pipeline slides, which includes all three phases. The control group read a set of abridged slides focused on recursion and loop-focused from a popular computer science textbook publishing company.

Unlike the first research study, the second research study contained an "interview" portion. The participants were asked to "talk-aloud" their answers to the interviewer. Talk-alouds, also sometimes called "think-alouds" have demonstrated promise in teaching computer science, as they help to clarify problem statements [23]. They also provide instructors/researchers with valuable feedback as to what goes through the mind of a student when solving a problem [24]. In this study, the interviewer periodically asked the participants questions about their thought process or interjected as a means of engaging with quieter participants.

In an attempt to remove as much experimenter-bias as possible, the submissions were evaluated/graded by a non-intervening third party, who has no relation or interest in the outcome of the study. Before any grading occurred, the authors blinded the submissions, where each participant was assigned an arbitrary letter. The third party did not know who participated in the study or was aware of which group any participant belonged to. To award and deduct points, the third party used a rubric created ahead of time (by the lead author). The third party graded two of the questions, while the lead author of this paper graded the other two.

b) Results: Each question was out of 50 points. Figure 4 plots the scores of the experimental group ($n = 8$) and the control group ($n = 7$). The legend to its right specifies the symbol for the respective group. The x-axis shows pre-test scores, and the y-axis shows post-test scores.

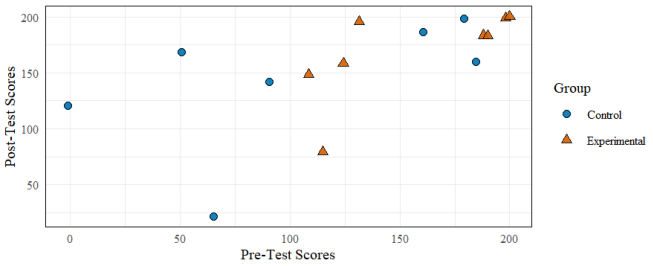

Fig. 4: Experiment 2: Pre-test vs Post-test Scores by Group

Running a Mann-Whitney U test of the difference of post and pre-test scores showed a lack of statistical significance between the two groups ($W = 34$, $p = 0.523$). We again attribute the lack of significance to the very small sample size.

In addition to the quantitative analysis, like the first study, we also examined the most common mistakes made by the participants of both groups. It was common for participants to outright give up on questions that they felt were too difficult. Below we list the common mistakes encountered by students in the post-test. (See Appendix A for the questions from the pre-test of experiment 2; see Appendix B for the questions from the post-test of experiment 2.)

For the `expt` question, mathematical mistakes were more prevalent than programming errors: multiplying the base by itself and omission of critical pieces (e.g., recursive calls, multiplications); 7/15 participants made mistakes. Off-by-one errors were surprisingly non-existent, but almost every participant had a different loop condition, or some variation thereof. For example, some started from $i = 0$ to b, $i = 1$ to $b + 1$, and so forth. What was more interesting to see was the variation in how the participants handle base cases. Some participants (4/15) superfluously used more than two clauses in their case analyses, with a separate one for handling when $b = 1$.

For the `collatz` question, no single issue was commonly encountered by the participants. Rather, mistakes were different across the board, ranging from invalid recursive call/updates (3/15), incorrect conditionals (1/15), no base cases (1/15), and no recursive calls (1/15).

It is easy to tell when an experimental participant directly used the translation pipeline. It is not as easy to tell when an experimental participant was *highly influenced* by the translation pipeline. We define highly influenced to mean that a participant used some portion of the translation pipeline in their answer. We hypothesize that 6 of the 8 experimental group participants were highly influenced by the translation pipeline, and 2 of the 8 definitively used the translation pipeline on at least one of the post-test problems.

To get an idea of how often participants spoke about relevant topics during the talk-aloud, we unitized and coded the post-test interview transcripts of each participant, placing a code category on each line of the transcript (note that a line is not necessarily a standalone sentence; it's a separation by the recording software). The goal of the qualitative analysis was to examine exactly *how* students write loops through utterances of their thought processes. The lead researcher poured through the transcripts of each participant, identifying key phrases and words related to the experiment. (Samples of codes include "Reference to base case", "Referencing accumulators ", "Referencing helper functions", "Use of tail recursion", "Discussion of what kind of loop to use", "Use of global variables", "Thinking out loud.")

We ran a Mann-Whitney U test of significance on the number of units coded. The test shows that, during the post-test talk-aloud, experimental group participants talk more about the questions than the control group participants ($W = 11, p = 0.055$). A correlation test between the two groups and the pre/post-test scores versus units coded is listed in Figure 5.

Group	Pre-test (r)	Post-test (r)
Experimental	0.677	0.581
Control	0.334	0.755

Fig. 5: Correlation Coefficient of Test Scores vs Units Coded

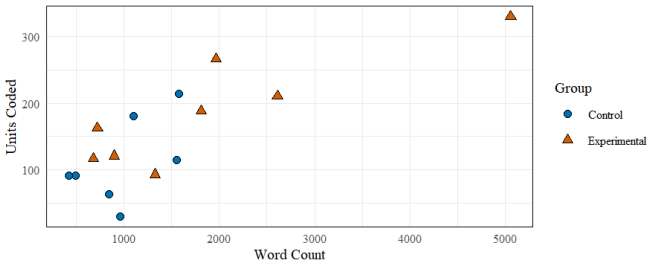

Fig. 6: Word Count vs Units Coded by Group

V. Discussion and Implications

In this section we will discuss and describe the results from both of our empirical research studies.

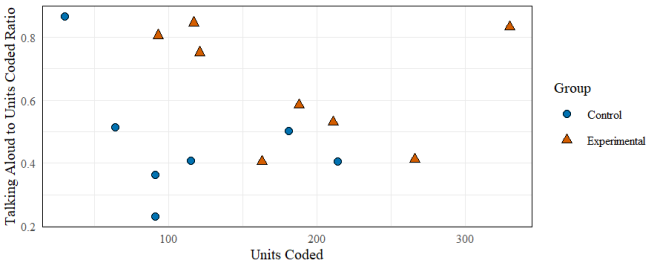

Fig. 7: Talk-Aloud Units Ratio by Group

A. Discussion of Research Study #1

Our first research question asks wether the translation pipeline helps students who understand conditionals and recursion to write loops. Given the relatively weak pre-test scores of the experimental group compared to their post-test scores (see Figure 3), we claim that our methodology prepared them for the loop conversion questions in the post-test.

The non-mechanical part of the translation pipeline, i.e., going from a problem statement to a recursive, then tail recursive solution, is conveyed by a presupposition of knowledge in designing recursive functions. Hence, while our efforts to filter out students who lack the necessary prerequisite knowledge were with good intentions, it is possible that participants volunteered without knowing the intensity at which their ability to interpret recursive code would be tested.

In general, we still perceive that the mechanical component of the translation pipeline has yielded successful results.

B. Discussion of Research Study #2

The overarching premise and motivation behind the second research study was to address the threats to validity of the first study. We wanted to ensure that students were exposed to all stages of the translation pipeline, not just those that are artificially cherry-picked. This approach ensures that we retain the original purpose of the translation pipeline: to help students write correct loops.

One area of controversial contention in CS1 discussions is the presentation order of topics [10], [20], [25]. As mentioned, some introductory computer science classes teach recursion before loops, whereas most others are the other way around, emphasizing imperative programming. Participants taught under the latter umbrella tended to struggle with the problem dynamics, particularly those in the experimental group. One participant remarked that they, "...had been taught loops before recursion, which is why [my] brain works that way." An experimental participant stated that, while the post-test was easier than the pre-test (which may suggest that the translation pipeline helped), designing the recursive functions were harder than writing loops.

A threat to validity that arises when working with students across various CS1 backgrounds is the language of choice. Our experiments all use the Python programming language. (The control group in the first study, however,

read through Java slides.) In the pre-screening survey, participants were required to verify their confidence in not only Python programming but also recursive problem-solving. The pre-screening survey contained two recursive tracing-based questions. Of course, such evaluations are self-imposed and may not transfer well to the pressure of what reduces to a Python programming interview, similar to real-world software engineering interviewees.

Topic presentation likely affected the ability and confidence of the participants. We observed that, while students in both groups preferred loops to recursion (based on their choice to write the loop solution before the one that uses recursion), those in the experimental group often swapped to writing the recursive algorithm first in the post-test. A participant, when questioned on what Collatz algorithm they would write first, responded with, "First...let's start with the recursive [solution] for this one." We are unsure whether these decisions are a result of presenting recursion as a translation pipeline *to* loops or if it is an effect of the problem selection. Exponentiation and the Collatz conjecture are naturally-recursive functions, versus the "iterative-inclination" (and comparatively harder structurally-recursive nature) of counting binary digits and finding the maximum of a list of integers.

We also noticed that, even if the experimental students did not exactly follow the translation pipeline, they read through it closer and constantly referred back to it whilst talking through their code. One experimental participant remarked, "...I know I didn't use [the translation pipeline], but I really did like it...it actually feels like things have a purpose...each part has a purpose that it plays in defining the function." Two participants questioned whether they were required to use the translation pipeline, i.e., if they had to follow it to the letter, so it seems clear that they were certainly influenced by the pipeline, even if they did not explicitly look at it much when writing their solutions to the post-test. During the post-test (and only during the post-test), another experimental participant continuously referenced the notion of "driver" and "helper" functions: two prominent components of the translation pipeline. It also stands to say that the experimental group materials could have been written in a clearer way, with one participant saying that, in regard to the post-test, "[I] should have asked if helper functions were allowed." Those participants in the control group, on the other hand, rarely referenced the provided slides, perhaps a nod to their lack of utility.

VI. Limitations & Future Work

In subsequent work we aim to potentially improve three aspects of the research design:

1) The translation pipeline,
2) How we teach the translation pipeline,
3) How we engage with participants in a research study.

We may view the current translation pipeline as insufficient, and if so, how can we change its design? We may wish to either add, remove, or alter existing steps for various reasons: complexity, time-to-complete, or importance to the task of writing loops. Moreover, if the existing pipeline suffers in quality, it is almost certain that how we teach it similarly suffers.

As is usual with participant-driven studies, obtaining enough participants to deduce a statistically-significant result in one way or the other is non-trivial.

We would also like to explore how a third alternative, e.g., no intervention, fares in comparison to the experimental and control group intervention tactics. An experiment with these data points may help to solidify a conclusion of whether students truly benefit from the translation pipeline. By this claim, we mean does the translation pipeline produce a noticeable difference among student performance when writing loops.

Another possible experiment might include removing the control group altogether, focusing entirely on the translation pipeline. Both experiments in this paper do not *enforce* experimental participants to use the translation pipeline in their solution. Instead, the two groups are encouraged to use what "they learned from the lecture material." A hypothetical experiment might have two groups: one where the translation pipeline is not "required," and another where we "enforce" its adoption. This kind of experiment may increase the number of subjects that "definitively used" the translation pipeline and, potentially, their scores on a post-test assessment.

Future improvements may also include a revision or further randomization of our selected test questions. The results from our qualitative analysis might suggest that either the pre-test or post-test fail to adequately measure success in writing loops as a consequence of their difficulty (or lack thereof). It is challenging to choose questions that do not alienate students with their intrinsic difficulty that also measure growth.

Finally, a hypothesis that is supported by our data is whether the translation pipeline helps students talk-aloud their problem-solving thought process; it provides students a language for talking about loops. While it may not immediately result in better performance/outcomes, it could be very interesting to see how people think about things. The talk-aloud interview strategy helped to unearth some of these questions, which pose the potential for future work.

VII. Conclusion

Computer science students struggle at several points in a curriculum. This paper pointed out one such instance in the design of loops. Particularly for those students who come from a recursive-driven background, the concept and implementation of loops can be a challenging hurdle to overcome.

We have proposed a pedagogically-motivated translation pipeline for students to mechanically convert tail recursive functions into iterative counterparts. We complemented this proposal with two empirical studies to gauge whether or not students (who already have a working understanding of conditionals and recursions) can correctly design loops. The first of our studies concentrated on pure translation from tail recursive functions and the other generalized the idea to the entirety of our translation pipeline. The studies together demonstrate that students appear have a healthier understanding of how to write

loops from problem statements after some application of the translation pipeline. We hope that future experimentation and a larger participant pool will further validate our hypotheses and pose new interesting questions.

Appendix A
Pre/Post-Test Questions in Experiment #1

The format of the questions in the first experiment received tail recursive implementations of functions and were tasked with converting them into equivalent loops.

Pre-test:

1) Return whether a given positive integer is prime.
2) Compute the sum of the digits of a given positive integer.

Post-test:

1) Sum the numbers of a list.
2) Naturally reverse the elements of a list.

Appendix B
Pre/Post-Test Questions in Experiment #2

Pre-test:

1) Given a binary string, return the number of `'1'`s.
2) Return the max number in a list of integers.

Post-test:

1) Given two integers a, b, return a^b naturally.
2) Return a comma-separated string of "Collatz" numbers from a given positive integer n to 1.

Acknowledgments

We would like to acknowledge the university where we conducted the study for their Amazon gift cards funding. Finally, we thank Tim Ransom, who helped grade the assessments from the second research study. The research studies performed in this paper were approved by the lead author's Internal Review Board (IRB), numbers #19959 and #24383.

References

[1] D. Wood, J. S. Bruner, and G. Ross, "The role of tutoring in problem solving*," *Journal of Child Psychology and Psychiatry*, vol. 17, no. 2, pp. 89–100, 1976. [Online]. Available: http://dx.doi.org/10.1111/j.1469-7610.1976.tb00381.x

[2] S. L. Gerhart, "Correctness-preserving program transformations," in *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '75. New York, NY, USA: Association for Computing Machinery, 1975, p. 54–66. [Online]. Available: https://doi.org/10.1145/512976.512983

[3] E. Schanzer, K. Fisler, S. Krishnamurthi, and M. Felleisen, "Transferring skills at solving word problems from computing to algebra through bootstrap," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE 2015, Kansas City, MO, USA, March 4-7, 2015*, A. Decker, K. Eiselt, C. Alphonce, and J. L. Tims, Eds. ACM, 2015, pp. 616–621. [Online]. Available: https://doi.org/10.1145/2676723.2677238

[4] S. Thorgeirsson, L. C. Lais, T. B. Weidmann, and Z. Su, "Recursion in secondary computer science education: A comparative study of visual programming approaches," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1321–1327. [Online]. Available: https://doi.org/10.1145/3626252.3630916

[5] H. Abelson, G. J. Sussman, and with Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge: MIT Press/McGraw-Hill, 1996.

[6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs*, 2nd ed. Cambridge: MIT Press, 2018. [Online]. Available: http://www.htdp.org/2018-01-06/Book/

[7] S. Tobin-Hochstadt and D. V. Horn, "From principles to practice with class in the first year," in *Proceedings Second Workshop on Trends in Functional Programming In Education, TFPIE 2013, Provo, Utah, USA, 13th May 2013*, ser. EPTCS, P. K. F. Hölzenspies, Ed., vol. 136, 2013, pp. 1–15. [Online]. Available: https://doi.org/10.4204/EPTCS.136.1

[8] M. Felleisen, M. Flatt, R. B. Findler, K. E. Gray, S. Krishnamurthi, and V. K. Proulx, "How to design classes," https://felleisen.org/matthias/HtDC/htdc.pdf, 2012.

[9] I. M. Santos, M. Hauswirth, and N. Nystrom, "Experiences in bridging from functional to object-oriented programming," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, ser. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 36–40. [Online]. Available: https://doi.org/10.1145/3358711.3361628

[10] F. A. Turbak, C. S. Royden, J. L. Stephan, and J. Herbst, "Teaching recursion before loops in cs1," 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:5603321

[11] S. Joosten, K. Berg, and G. Hoeven, "Teaching functional programming to first-year students," *Journal of Functional Programming*, vol. 3, pp. 49–65, 01 1993.

[12] K.-C. Lu, S. Krishnamurthi, K. Fisler, and E. Tshukudu, "What happens when students switch (functional) languages (experience report)," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, aug 2023. [Online]. Available: https://doi.org/10.1145/3607857

[13] E. Makri, N. Choudhary, and C. H. Muntean, "Computer programming: A case study of teaching loop statement by using an interactive educational game," *International Journal for Digital Society*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:214242041

[14] M. Tr, A. Bocevska, I. Nedelkovski, and S. Savoska, "Game theme based instructional module to teach loops and choice statements in computer science courses," *Annual of Sofia University St. Kliment Ohridski. Faculty of Mathematics and Informatics*, vol. 110, pp. 127–138, 11 2023.

[15] A. Gomes, W. Ke, C.-T. Lam, A. R. Teixeira, F. B. Correia, M. J. Marcelino, and A. J. Mendes, "Understanding loops: a visual methodology," in *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, 2019, pp. 1–7.

[16] I. Cetin, "Teaching loops concept through visualization construction," *Informatics in Education*, vol. 19, pp. 589–609, 12 2020.

[17] M. T. Morazán, "How to design while loops," *arXiv [cs.OH]*, 2020.

[18] L. E. M. Briana B. Morrison and A. Decker, "The curious case of loops," *Computer Science Education*, vol. 30, no. 2, pp. 127–154, 2020.

[19] R. Catrambone, "The subgoal learning model: Creating better examples so that students can solve novel problems," *Journal of Experimental Psychology: General*, vol. 127, no. 4, pp. 355–376, Dec. 1998.

[20] C. M. Kessler and J. R. Anderson, "Learning flow of control: Recursive and iterative procedures," *Human–Computer Interaction*, vol. 2, no. 2, pp. 135–166, 1986.

[21] S. Wiedenbeck, "Learning iteration and recursion from examples," *International Journal of Man-Machine Studies*, vol. 30, no. 1, pp. 1–22, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737389800185

[22] G. J. Sussman and G. L. Steele, "SCHEME: An Interpreter for Extended Lambda Calculus — hdl.handle.net," http://hdl.handle.net/1721.1/5794, 12 1975.

[23] N. Arshad, "Teaching programming and problem solving to cs2 students using think-alouds," *SIGCSE Bull.*, vol. 41, no. 1, p. 372–376, Mar. 2009. [Online]. Available: https://doi.org/10.1145/1539024.1508998

[24] K. A. Ericsson and H. A. Simon, "Verbal reports as data," *Psychological Review*, vol. 87, no. 3, pp. 215–251, 1980. [Online]. Available: https://doi.org/10.1037/0033-295X.87.3.215

[25] R. M. Siegfried, K. G. Herbert-Berger, K. Leune, and J. P. Siegfried, "Trends of commonly used programming languages in cs1 and cs2 learning," in *2021 16th International Conference on Computer Science & Education (ICCSE)*, 2021, pp. 407–412.