How to Design Loops

Joshua Crotts [1] Chung-chieh Shan [1] Sam Tobin-Hochstadt [1]

[1]Indiana University Bloomington

Abstract

Loops in programming languages are difficult for beginning students to construct correctly. 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.

Thus, we introduce a step-by-step process for loop construction that beginning programming students can follow. Our approach begins with a recursive function, and provides a translation pipeline resulting in a correct loop implementation.

We focus on students who have prior experience with recursive functions, since this provides additional challenges for teaching loops.

We perform an evaluative study on students from the CS1 and CS2 population to test our methodology against the traditional approach to teaching loops in our CS2-level course.

Our results indicate that our translation pipeline method improves students' success in writing loops.

Introduction and Research Questions

Students learn how to write loops in many computer science classes, which often leads to confusion for many reasons:

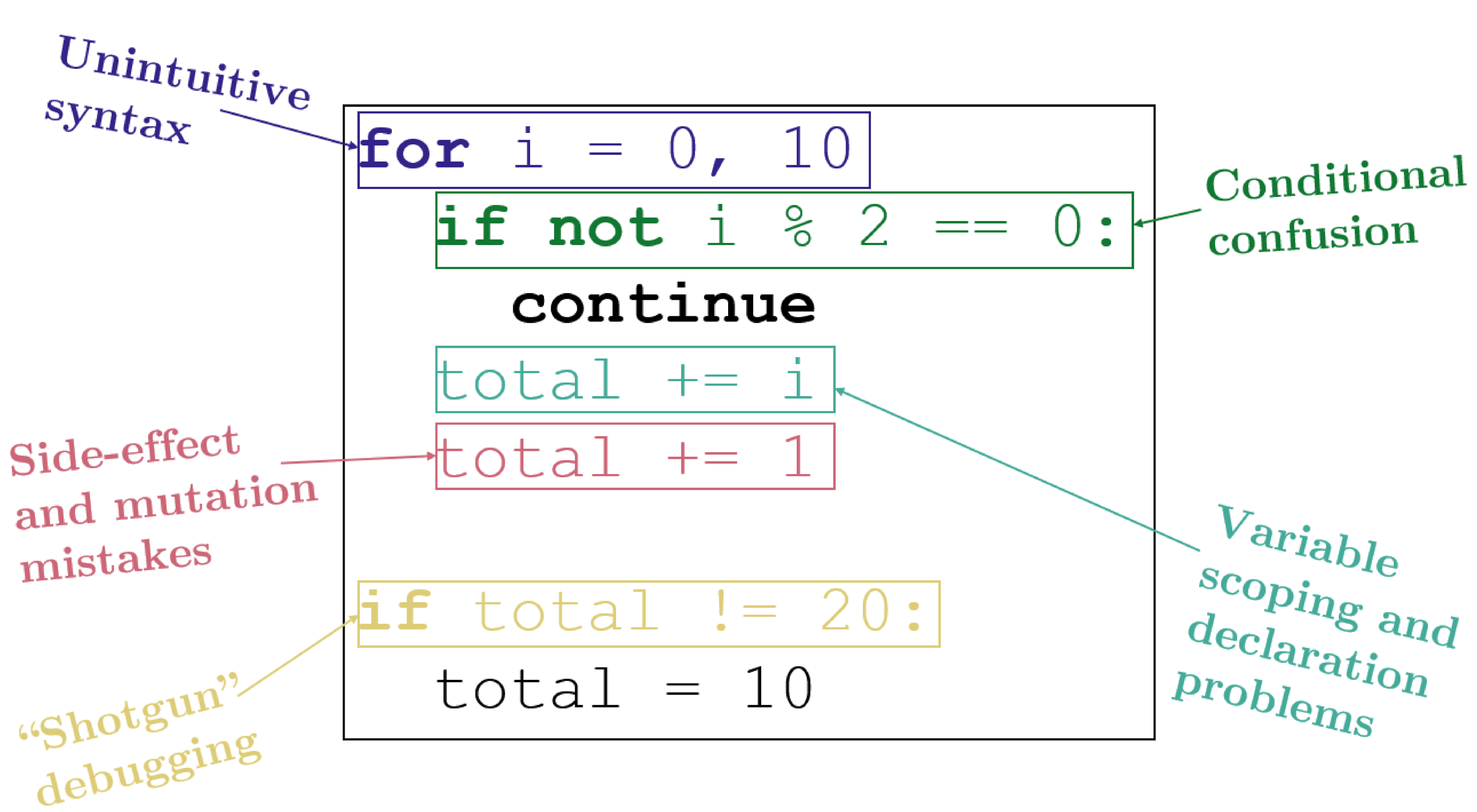

Figure 1. Sample Answer With Mistakes to Prompt, "Write a loop to sum the even numbers from 1 to 10."[2]

We propose such a scaffolded approach via a translation pipeline from a *tail recursive* function to a loop, with the following research questions:

RQ1: Does the mechanical part of 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?

This answer is a reproduction of an amalgamation of multiple errors that many students make and have made during instructor observation. It is not an authentic student solution.

Methods

There are 7 steps to the translation pipeline from tail recursion to loops:

1. Without worrying yet about using a loop, the student first designs a recursive function that solves the problem.
2. The student checks if the function just designed is *tail recursive.*
3. 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),
   iii. each updated parameter in a recursive call.
4. The student converts the accumulators into local variables at the top of the loop definition. These are marked with iterative variable purpose statements.
5. The student writes the **while** keyword, followed by the logical negation of our base case conditions via the ! operator.
6. The student turns updated parameters into assignment statements within the loop body.
7. Lastly, the student adds a **return** statement to return each non-recursive return value of the tail recursive function.

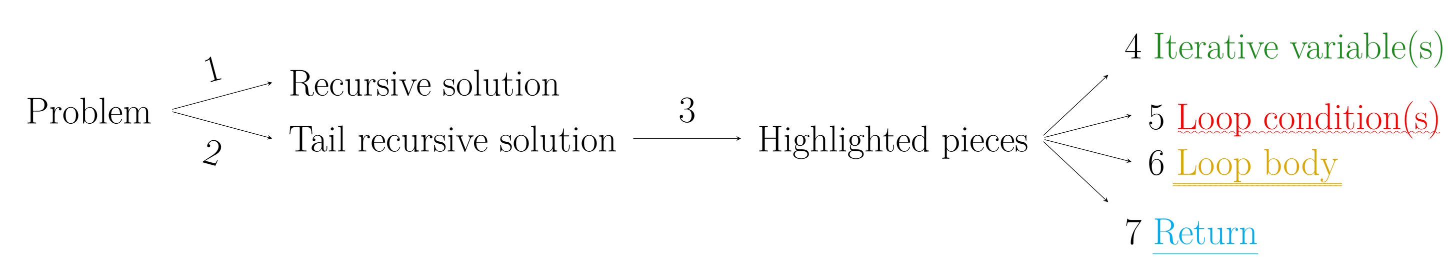

Figure 2. General Translation Pipeline

Results

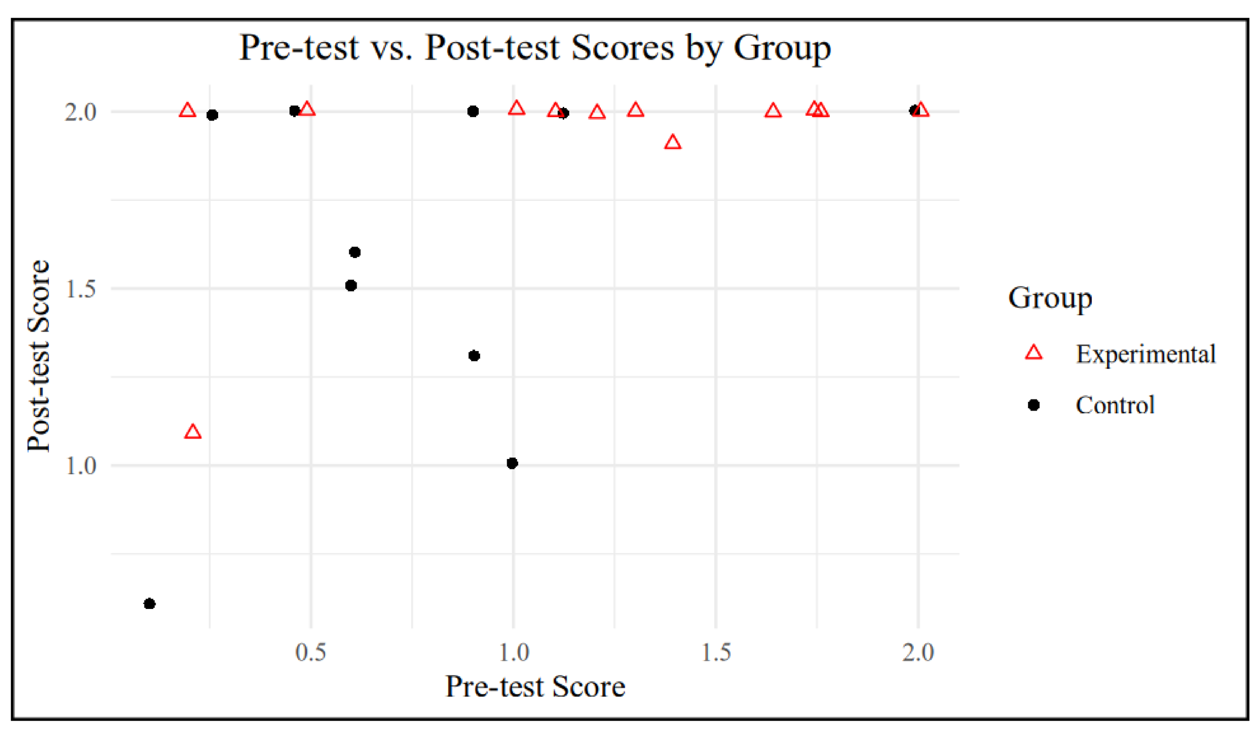

Figure 3. Pre-test vs Post-test Scores by Group

Participants were scored on a pre-defined rubric out of 2 points per test. All of the experimental participants showed a sharp growth, whereas control participants showed less.

Our preliminary translation pipeline results are promising, and we plan to further improve either parts of the translation pipeline or the study design.

Empirical Evaluation

Participants were asked to take a pre-test on designing loops, read slide material, then take a post-test to demonstrate growth.

- The *experimental group* received print-out slides of designing loops via the translation pipeline.
- The *control group* were instructed to read through a set of slides about loops from a popular introductory Java programming textbook.

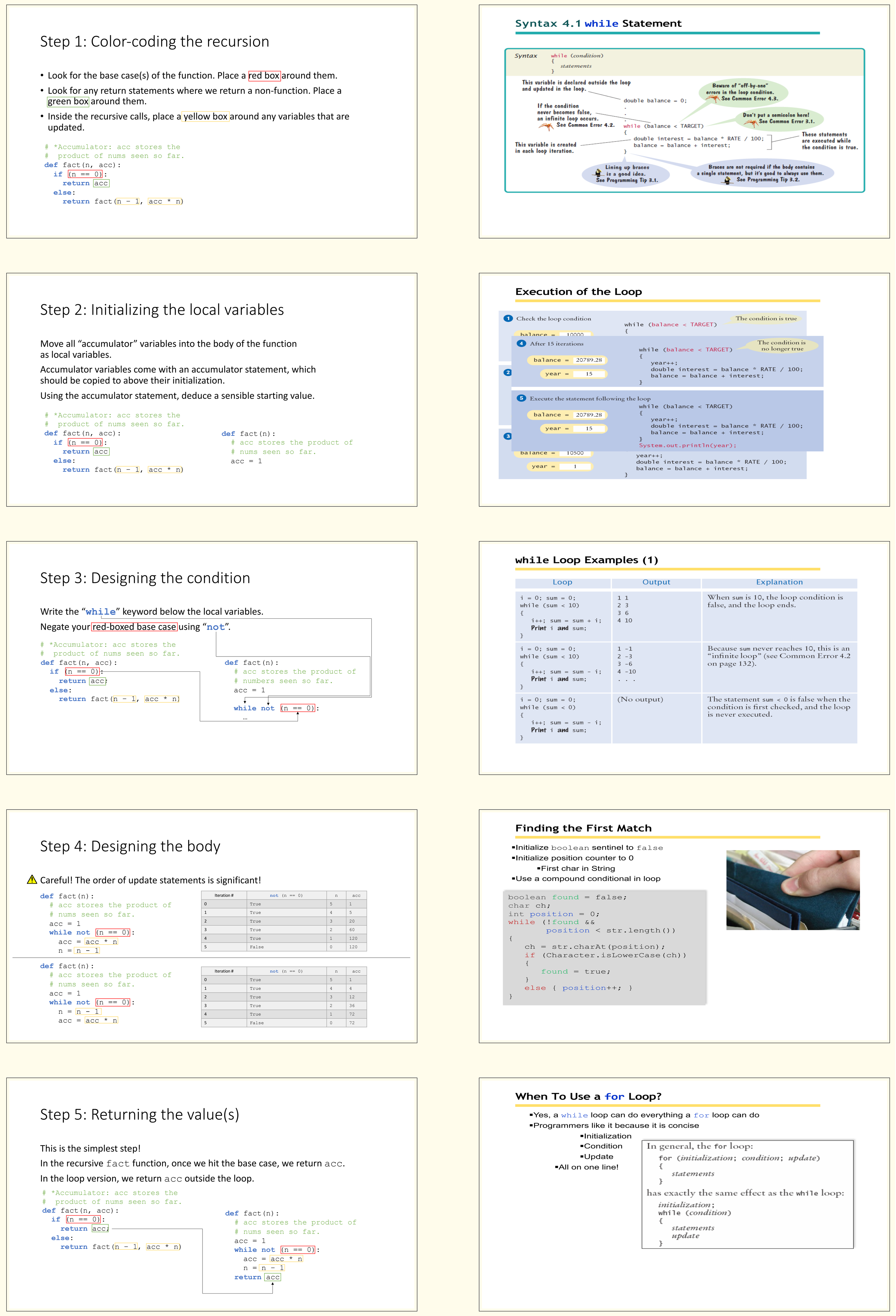

Figure 4. (Left) Experimental Group Slide Samples, (Right) Control Group Slide Samples

https://citl.indiana.edu/news/feature/celebration-of-teaching-2024.html

Indiana University Celebration of Teaching 2024, Bloomington, Indiana