

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:

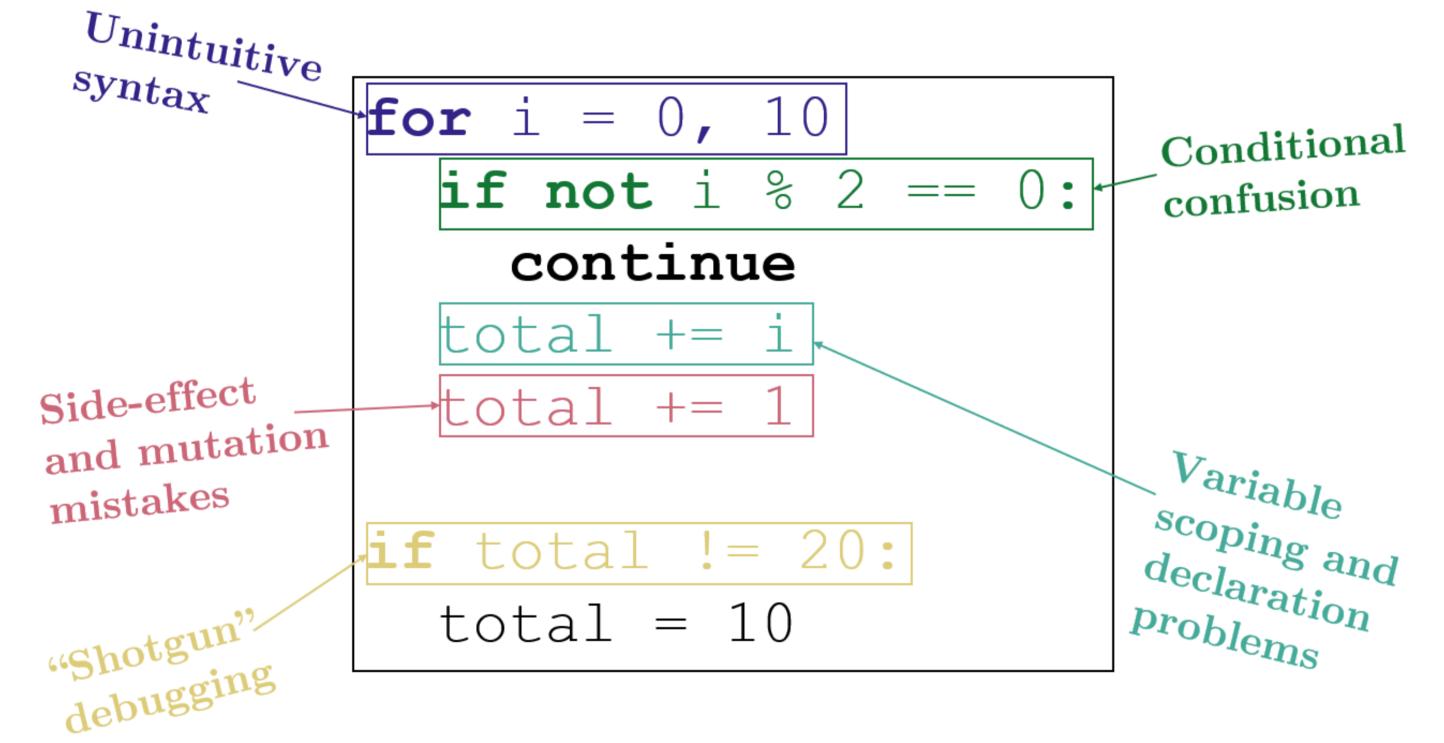


Figure 1. Sample Answer With Mistakes to Prompt, "Write a loop to sum the even numbers from 1 to $10."^2$

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.

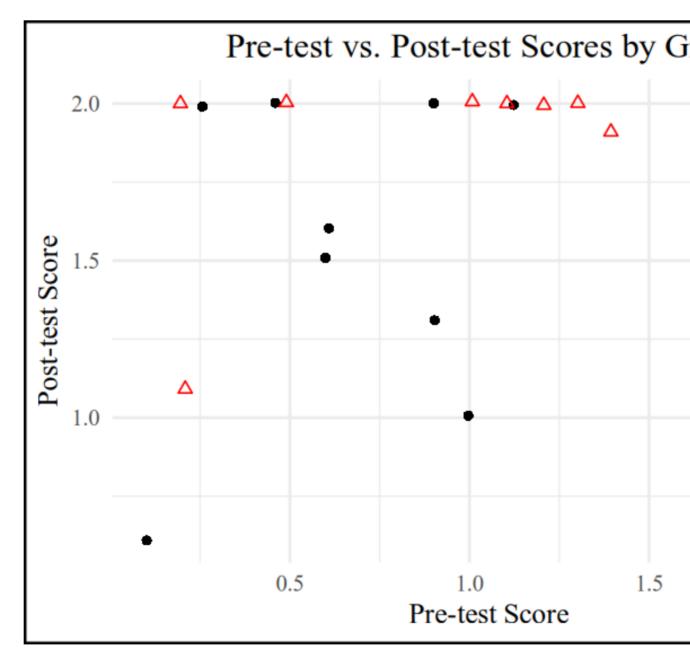
https://citl.indiana.edu/news/feature/celebration-of-teaching-2024.html

How to Design Loops Joshua Crotts¹ Chung-chieh Shan¹ Sam Tobin-Hochstadt¹ ¹Indiana University Bloomington

Methods

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

- then take a post-test to demonstrate growth. tion that solves the problem. lation pipeline. a popular introductory Java programming textbook. i. the base case condition(s), ii. the non-recursive return value(s), Step 1: Color-coding the recursion iii. each updated parameter in a recursive call. Look for any return statements where we return a non-function. Place a • Inside the recursive calls, place a yellow box around any variables that are definition. These are marked with iterative variable purpose statements. updated. # *Accumulator: acc stores the # product of nums seen so fa def fact(n, acc): **if** (n == 0): return acc <u>case conditions</u> via the ! operator. else: return fact(n - 1, acc * n) body. Step 2: Initializing the local variables value of the tail recursive function. Move all "accumulator" variables into the body of the function as local variables. Accumulator variables come with an accumulator statement, which should be copied to above their initialization 4 Iterative variable(s) Using the accumulator statement, deduce a sensible starting value # *Accumulator: acc stores th Recursive solution # product of nums seen so far Problem \rightarrow 5 Loop condition(s) def fact(n, acc): def fact(n): **if** (n == 0): # acc stores the product of Highlighted pieces Tail recursive solution return acc # nums seen so far. else: acc = 1 ~ 6 Loop body return fact(n - 1, acc * n) 7 Return Figure 2. General Translation Pipeline Step 3: Designing the condition Write the "while" keyword below the local variables Negate your red-boxed base case using "not". # *Accumulator: acc stores the # product of nums seen so fat Results def fact(n, acc): def fact(n): **if** (n == 0): # acc stores the product return acc; # numbers seen so far acc = 1else: return fact(n - 1, acc * n) while not (n == 0): Pre-test vs. Post-test Scores by Group 2.0 $\triangle \Delta$ Step 4: Designing the body Careful! The order of update statements is significant! Group def fact(n): # acc stores the product of # nums seen so far. ▲ Experimental ۰ acc = 1 while not (n == 0) acc = acc * n Control n = n - 1 Δ def fact(n): # acc stores the product # nums seen so far. acc = 1while not (n == 0) n = n - 1 acc = <mark>acc * n</mark> 0.5 1.5 2.0 1.0 Pre-test Score Step 5: Returning the value(s) This is the simplest step! In the recursive fact function, once we hit the base case, we return acc Figure 3. Pre-test vs Post-test Scores by Group In the loop version, we return acc outside the loop # *Accumulator: acc stores the # product of nums seen so far
- 1. Without worrying yet about using a loop, the student first designs a recursive func-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: 4. The student converts the accumulators into local variables at the top of the loop 5. The student writes the **while** keyword, followed by the logical negation of our <u>base</u> 6. The student turns updated parameters into assignment statements within the loop 7. Lastly, the student adds a **return** statement to return each non-recursive return



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.

Conditional
confusion

Group Slide Samples

lef fact(n):

acc = 1

n = n - 1 return acc

nums seen so fa

while not (n == 0) acc = acc * n

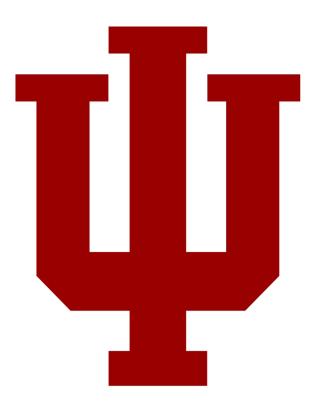
lef fact(n, acc)

return acc

return fact(n - 1, acc * n)

if (n == 0):

else:



Empirical Evaluation

- Participants were asked to take a pre-test on designing loops, read slide material,
- The *experimental group* received print-out slides of designing loops via the trans-

• The *control group* were instructed to read through a set of slides about loops from

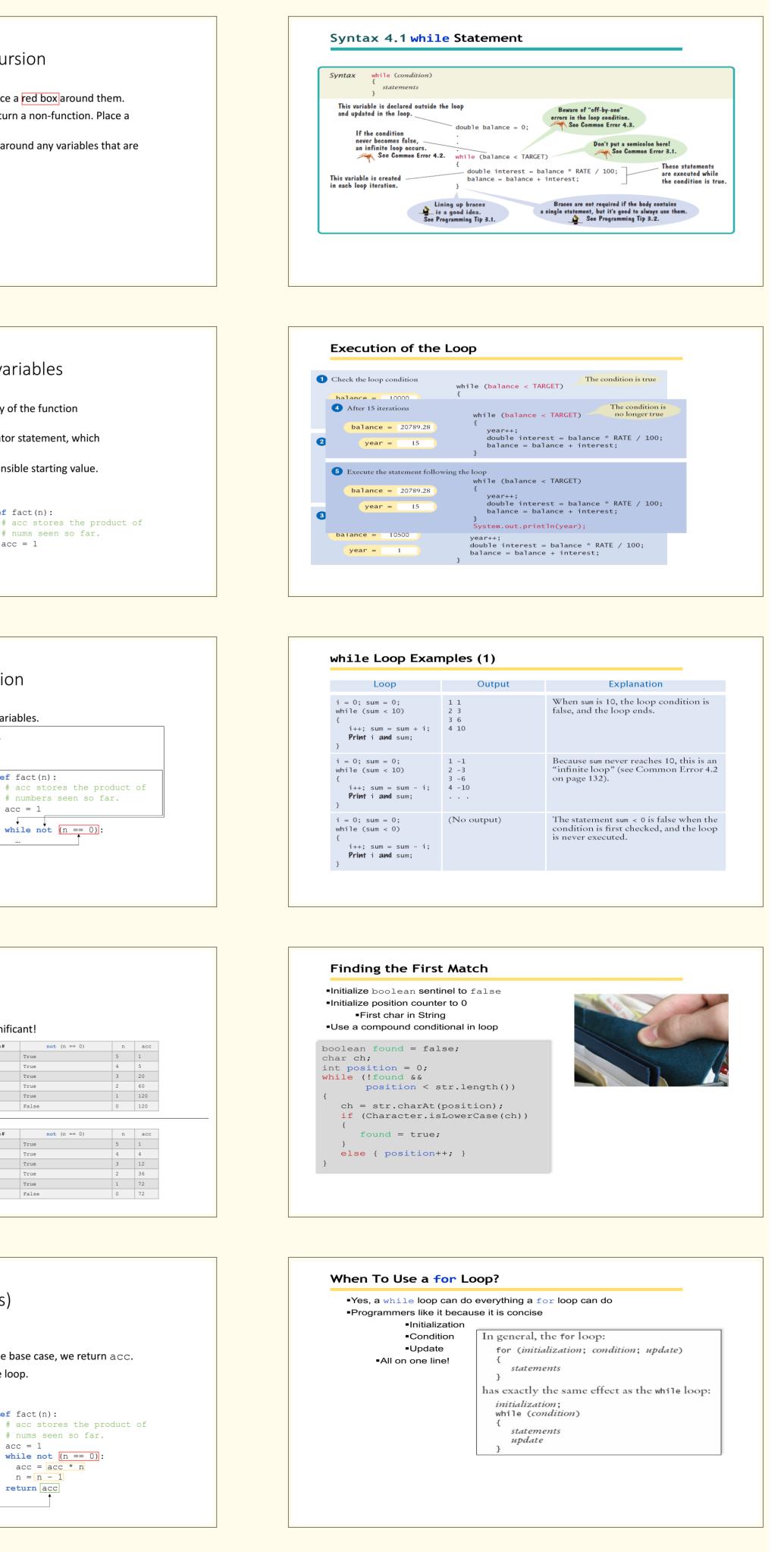


Figure 4. (Left) Experimental Group Slide Samples, (Right) Control