Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent

Mohammed Hassan
University of Illinois
Urbana, United States
mhassan3@illinois.edu

Kathryn Cunningham
University of Illinois
Urbana, United States
katcun@illinois.edu

Craig Zilles
University of Illinois
Urbana, United States
zilles@illinois.edu

ABSTRACT

Background and context. "Explain in Plain English" (EiPE) questions ask students to explain the high-level purpose of code, requiring them to understand the macrostructure of the program's intent. A lot is known about techniques that experts use to comprehend code, but less is known about how we should teach novices to develop this capability.

Objective. Identify techniques that can be taught to students to assist them in developing their ability to comprehend code and contribute to the body of knowledge of how novices develop their code comprehension skills.

Method. We developed interventions that could be taught to novices motivated by previous research about how experts comprehend code: prompting students to identify beacons, identify the role of variables, tracing, and abstract tracing. We conducted think-aloud interviews of introductory programming students solving EiPE questions, varying which interventions each student was taught. Some participants were interviewed multiple times throughout the semester to observe any changes in behavior over time.

Findings. Identifying beacons and the name of variable roles were rarely helpful, as they did not encourage students to integrate their understanding of that piece in relation to other lines of code. However, prompting students to explain each variable's purpose helped them focus on useful subsets of the code, which helped manage cognitive load. Tracing was helpful when students incorrectly recognized common programming patterns or made mistakes comprehending syntax (text-surface). Prompting students to pick inputs that potentially contradicted their current understanding of the code was found to be a simple approach to them effectively selecting inputs to trace. Abstract tracing helped students see high-level, functional relationships between variables. In addition, we observed student spontaneously sketching algorithmic visualizations that similarly helped them see relationships between variables.

Implications. Because students can get stuck at many points in the process of code comprehension, there seems to be no silver bullet technique that helps in every circumstance. Instead, effective instruction for code comprehension will likely involve teaching a collection of techniques. In addition to these techniques, meta-knowledge about when to apply each technique will need to be learned, but that is left for future research. At present, we recommend teaching a bottom-up, concrete-to-abstract approach.

ICER '23 V1, August 07–11, 2023, Chicago, IL, USA

ACM ISBN 978-1-4503-9976-0/23/08…$15.00
https://doi.org/10.1145/3568813.3600140

CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

KEYWORDS

tracing, CS 1, program comprehension

ACM Reference Format:
Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. 2023. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1 (ICER '23 V1), August 07–11, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3568813.3600140

1 INTRODUCTION

Skill in program comprehension is undoubtedly necessary to be a strong programmer. "Explain in Plain English" (EiPE) questions capture an important aspect of students' program comprehension ability: the capacity to identify the overall *purpose* or *intent* of a piece of code [18]. This competency to explain code is not only useful on its own—it also been consistently found to correlate with other programming skills, like code tracing and writing [9, 23, 25], suggesting that similar knowledge underlies all of these abilities.

We know a great deal about the various techniques that experts use when understanding code. Typically programmers apply a mix of top-down and bottom-up comprehension strategies depending on what is most beneficial in their situation (e.g., top-down if the code is familiar) [21, 24, 31, 60]. A wide variety of models have been proposed for program comprehension [4, 21, 24, 31, 33, 49, 60]. What is common among these comprehension strategies is that they involve some process of decomposing a large piece of code into smaller sub-parts alongside the recognition of key code pieces (known as "beacons") that allow the programmer to make inferences throughout the decomposition process.

However, less is known about how novices develop the ability to understand code. Some observational studies have noted the techniques novices use when trying to understand code [57]. These studies found that novices' ability to trace code became more abstract over time, meaning that novices could describe code in larger pieces and trace without using specific values. The block model has been proposed as a means of characterizing a student's progress towards developing the ability to relate small pieces of code to the overall purpose of code [44]. Despite all we know about expert code

comprehension, few studies have explored what should be taught to students to help them learn code comprehension.

In this study, we ask:

> *What techniques can be taught to students to help them be successful at code comprehension tasks like Explain in Plain English problems?*

To investigate this research question, we designed four interventions based on the activities that have been observed when experts and novices successfully understand code. They include: (1) recognition of beacons, (2) understanding the role of variables, (3) (concrete) tracing, and (4) abstract tracing. Section 2 describes the research that motivates these interventions. We did not know a priori which of these techniques could be learned by novices and in what circumstances they would aid their understanding of code.

To study these interventions, we conducted a series of "think aloud" interviews that occurred during and shortly after students participation in an introductory programming course. The interviews were recorded and transcribed and analyzed qualitatively. Our methods are discussed in Section 3.

Based on our analysis of the data, we found it useful to categorize the interventions into two clusters. The first cluster included interventions (beacons, role of variables) that primarily helped novices decompose programs into useful pieces. The second cluster included interventions (concrete tracing, abstract tracing) that helped novices resolve mistakes from reading syntax and understand relationships between parts of the code. As such, our results are presented in two sections, Sections 4 and 5, respectively. In Section 6, we synthesize our findings into suggestions about how code comprehension should perhaps be taught.

2 BACKGROUND

As noted in the introduction, expert programmers tend to use a mix of top-down and bottom-up strategies for code comprehension. The bottom-up model [33] is commonly applied when the code initially seems unfamiliar to the programmer. This model consists of a program model and a situation model. The program model is a control-flow mental representation of the program, made by grouping chunks of the code and leveraging beacons (discussed below) throughout this process (e.g., identifiers, comments, cues to common patterns or plans). Afterward, they build a situation model, a data-flow mental representation of the program, also built by grouping chunks, where programmers apply their problem domain knowledge.

In Brooks' top-down model [4], the programmer begins with an initial hypothesis of the purpose of the entire program based on their domain knowledge (real-world knowledge of the problem the program aims to solve) and hierarchically creates subsidiary hypotheses to explore specific, implementation-level details needed to verify the parent hypothesis leveraging beacons throughout the process. Meanwhile, Soloway et al.'s top-down model involves recognizing code fragments (common patterns or plans) that seem familiar [49].

2.1 Plans & Plan Composition

Programming plans are equally important for program comprehension and program construction. Plans are 'canned' solutions representing chunks of knowledge that could be incorporated in various contexts [49]. Various standard elementary plans are well-known in the literature [37], such as initializing variables, averaging, swapping, counter-controlled loops, summing, and counting plans. Expert programmers have extensive knowledge of programming plans from prior experience and would apply and merge various plans when they write programs [48]. Experts use their knowledge of plans (or organized schemas of general pattern knowledge) to help them decompose problems into useful sub-goals. On the other hand, novices lack plan knowledge and thus do not know where to start to decompose problems and fail to find useful sub-goals within programs. Prior work argues that instructors should teach programming plans explicitly to students [66]. When instructors taught novices to use programming plans, they applied plans more often than a control group, performed better solving the code-writing questions, and were more confident in their understanding of their code-writing solutions [37].

Applying, nesting, and merging a set of programming plans to write code can be considered a method toward problem decomposition [48]. Students often fail to solve problems because they fail to take a problem description, break it down into sub-problems, implement the sub-problems, then compose the sub-problems together to the solution [29]. Issues of problem decomposition are partly due to students' poor understanding of programming semantics [29] and struggles to compose and merge sub-plans [52, 53, 61]. Merging (interleaving) plans introduce element interactivity, which is cognitively demanding for students [19, 47, 61, 63]. For example, a great cognitive burden exists when considering loop plans within programs and where to place them. Students also struggle to tailor (modify and adapt) plans to their needs [61]. Sometimes, students unproductively memorize plans rather than have a deep understanding of them [61].

2.2 Beacons

Beacons are commonly regarded as information-rich hints toward understanding the program's function [4]. Beacons can take on many forms, such as operations done on variables pertaining to design patterns [18], informative identifiers like variable names or function names [10], comments, focal segments such as sub-plans that serve the central functionality within complex plans [64], and any surface features that indicate the presence of recognizable operations, structures, or plans [4]. Expert programmers intently look for beacons when reading code [65]. Conversely, novices tend to read code more linearly, like natural language text, going from left to right, top to bottom and are often less successful in comprehending programs [5].

2.3 Role of Variables

The operations done on variables can pertain to design patterns and are integral parts of plans [18]. The *role* of variables are common, stereotypical operations that often occur in programs [40]. Ten roles of variables are known to cover 99% of novice-level procedural operations: fixed values, stepper, follower, most-recent holder, most wanted holder, gatherer, transformation, one-way flag, temporary, and organizer [39]. Prior work found that explicitly teaching students the role of variables increased students' programming

skills by integrating roles within the code they wrote [6, 42]. In one study, computer science educators learned the role of variables within an hour and could correctly identified the role of variables within code they read [1]. Students who were taught the role of variables performed better at explaining the lifecycle of the variable to peers and instructors, scaled by SOLO's taxonomy [46].

2.4 (Concrete) Tracing

Code tracing is the act of executing code either mentally or through bookkeeping on some media, which is commonly evaluated through "find the output" or "find the outcome" of code problems [30]. Being able to trace code does not necessarily indicate that a student has a high-level understanding of the purpose of the code [22]. Teague et al. [55] found that some students use tracing to understand code through inductive reasoning, where students would trace code multiple times and identifying patterns in the input-output pairs [54]. In this paper, we will refer to this technique as *concrete tracing* to help distinguish it from abstract tracing.

2.5 Abstract Tracing

Abstractly tracing a program is tracing without specific values but instead using a symbolic representation, sets, or ranges to represent multiple possible executions simultaneously. Teague et al. [54] observed students abstractly tracing code when solving EipE questions, where they would deductively reason about variables based on constraints rather than concrete values. These constraints pertain to the relationship between variables (e.g., `y = z + 11`) and can be based on conditional statements (e.g., `if x > 2`). Prior work has investigated using assertions [3, 8], tracing with symbolic values, design-by-contract assertions, and loop invariants to get students to reason about code symbolically [8]. Izu et al. [17] found that students deductively reason about code when solving reversibility problems, which ask students if a program's output can be restored to its original input state.

2.6 Block Model

The Block Model distinguishes between program comprehension tasks pertaining to understanding: 1) text-surface (syntactical *structure*), 2) execution behavior (algorithmic *structure*), and 3) purpose (intent, *function*) [18]. A program's intent is concerned with understanding why a programmer has written a program (i.e., external context, domain), which is the extrinsic purpose of the code and is qualitatively different from understanding execution behavior [32]. An example of a task pertaining to understanding the *structural* relationships between subsets of the code in the *program execution* dimension of the block model can be tracing code for a particular input to develop an understanding of the execution-state relationship between caller code and called procedural units. An example of a task pertaining to understanding the algorithm of the whole program at a high level (i.e., the macrostructure of *program execution* dimension) can be identifying a comprehensive set of inputs to access all control-flow paths of a program [18]. As for understanding *intent*, an example of such a task could be selecting suitable variable names within a program. This process involves understanding the *functional* relationship between (purposeful) sub-goals within the code. These relationships come together to establish the program's overarching goal or *intent* of the program. Choosing an appropriate name for the whole program pertains to understanding the macrostructure of the program's *intent* [18].

3 METHODS

3.1 Participants & Data Collection

The first author conducted and recorded 42 think-aloud interviews of introductory programming students solving EiPE questions over Zoom. These participants were undergraduate students who had taken or were in the process of taking a Python-based introductory programming course for non-technical majors at a large public U.S. university. These participants were 20 men and 22 women of traditional college age. With the institutional review board (IRB) and the course instructor's permission, we emailed the class roster during the Fall 2022 semester to recruit participants. The authors were not instructors or teaching assistants of the class and had no relationship to the participants. Fourteen of these participants were interviewed multiple times throughout the semester to observe any changes in behavior over time. These longitudinal participants were interviewed once near the beginning of the semester (3rd week), once during the middle of the semester (7th-9th week), and once toward the end of the semester (14th+ week).

During the interviews, participants were asked to complete multiple EiPE questions. These EiPE questions were written based on combinations of common patterns [16]. The EiPE questions did not have meaningful identifiers (e.g., meaningful variable or function names), as we wanted the students to infer the purpose of code based on execution behavior rather than simply reading the identifiers. Students were prompted to use an intervention strategy only when they were stuck solving the problem independently. Each participant was assigned to one of the four intervention conditions, discussed in Section 3.3 and below. In cases where the intervention was ineffective, an alternative intervention was sometimes used on the same problem, or the interviewer attempted a newly invented intervention.

3.2 Analysis Process

The think-aloud interviews were transcribed and analyzed in conjunction with video footage using an inductive coding approach. Our study focused on instances where interventions appeared to aid students in correctly explaining the code. Even in cases where interventions seemed ineffective, we observed additional actions by participants or the interviewer that facilitated correct code explanations.

First, the first author identified relevant areas of the 42 transcripts for later analysis. This was done by segmenting the instances when an intervention was used during the interviews and labeling that use of the intervention as successful or unsuccessful. Categories like 'beacons fail,' 'role of variables success,' 'tracing fail,' and 'tracing success' were used. Newly emerged interventions were also coded by the first author based on whether they successfully aided the students' code explanation. These segments of the transcripts were then organized into categories of each intervention's successful and unsuccessful uses.

Next, the first author conducted line-by-line inductive open coding of each intervention's successful instances. Since beacons consistently failed to aid problem-solving, the first author additionally coded its unsuccessful instances. This involved coding each transcript line to describe the action performed by the student, chunks of the code the student appeared to understand, and potentially the relationships between them. For example, codes included 'tracing with a focus on one variable' for the role of variables, 'resolve incorrect comprehension of syntax' or 'choose input to confirm false assumptions' for tracing, 'describe the range of all possible values of variable x' for abstract tracing, and 'describe dependency between variable x and y' for visualization and abstract tracing.

After the first author independently analyzed the entire dataset, he selected specific interview segments from each intervention for further discussion with the second and third authors, due to the sheer amount of data (42 interviews). All three authors reviewed these instances across 10 weeks.

To ensure the reliability of interpretations, the third author independently and inductively coded the selected data subset using transcriptions and video footage for his analysis. The 3rd author analyzed this subset blind to the 1st author's interpretation, independently generating his interpretations. Subsequently, all authors convened to reconcile their interpretations, discuss emerging themes, and agree on grouping interventions based on how they helped students solve the problem. For example, 'beacons' and 'role of variables' were grouped based on their effectiveness in dividing code into sub-problems.

3.3 Beacons Intervention

In this intervention condition (n = 11), we aimed to get students to try to read code more like experts by focusing on lines of code central to the program's functionality. To do this, we asked students to identify and explain a line that is "most important to the code." In our problems, the most information-rich lines are typically integral parts of a sub-plan occurring within the code (e.g., if-statement within a filter plan), where the sub-plan performs an operation central to the code's purpose [64]. We asked students to identify one line of code because our problems are short (e.g., only 4-20 lines). The hope was that students would independently integrate their understanding of that identified line into the sub-plan it pertains to and then the whole program.

3.4 Role of Variables Intervention

In this intervention condition (n = 11), we asked students about the roles of individual variables in the program. This was done in two ways. At the beginning of each session, longitudinal participants were taught the roles of variables [20, 40] that they had sufficient syntactic knowledge to understand. The order the roles were taught was similar to the best practices described by Kuittinen et al. [20]: In the first interview, we began with the simpler roles *fixed value*, *most recent holder*, and *temporary*. In the second interview, which was after the instructor of the course introduced loops, we taught *stepper*, *gatherer*, and the remaining roles. Whenever these participants struggled to understand the program, we asked them to identify the role of variables and then re-attempt the problem.

Later in our study, based on our observations of this first approach, we switched our intervention to ask participants to explain "the purpose of every variable," in addition to identifying the role of every variable, as we found this to increase the benefit of the intervention. For non-longitudinal participants in this condition, we did not teach roles of variables and, instead, simply asked participants to explain the purpose of every variable in the program.

3.5 Concrete Tracing Intervention

In this intervention condition (n = 10), we asked students to perform a concrete trace of the program when they appeared stuck solving the problem or repeatedly solved it incorrectly. We asked students to "try inputs into the program". If students did not choose inputs that sufficiently helped them understand the problem, the interviewer would guide them to select helpful inputs. Students were directed to bookkeep on paper or on the computer and not attempt to perform the trace in their head.

3.6 Abstract Tracing Intervention

In this intervention condition (n = 10), we taught students an abstract tracing approach where they note down the value of any unknown variable as a symbol. Each symbol has a corresponding data type (e.g., integer) and a range of values (e.g., any integer from negative to positive infinity). Arithmetic operations performed on these symbols are represented by arithmetic expressions rather than accumulated values (e.g., $E1 + E2 + E3$ for the sum of three integers).

In some cases, we prompted students to abstractly trace with a narrow range of inputs (1-3 for values from 1 to 3) rather than a representation for all possible values. Like the symbolic approach, they were directed to not simplify arithmetic expressions involving concrete values (e.g., 1 + 2 + 3 rather than 6). In both approaches, students were directed to bookkeep on paper or on the computer and not attempt to perform the trace in their head.

4 RESULTS: HOW INTERVENTIONS SUPPORTED NOVICES' PROBLEM DECOMPOSITION

When solving EiPE problems, we noticed that many of our participants initially attempted to comprehend the entire code without breaking up the problem into sub-problems. These participants appeared overwhelmed, likely due to the excessive cognitive load, as one student noted:

> It was a lot to try and wrap my head around with just looking at it, mentally I was trying to think about too much at once.

These participants appeared to struggle to understand the code by reading linearly, from top to bottom, left to right, likely because most arbitrary groups of adjacent lines of code do not have a complete purpose. As a result, an attempt to make sense of the code a few contiguous lines at a time, working from top to bottom, would be unlikely to reduce cognitive load because adjacent lines cannot always be effectively chunked together.

We found that two interventions, the beacons intervention and the role of variables intervention, sometimes helped participants

Assume that the variable `x` is a `string`.

```
def f3(x):
    e = "aeiou"
    for c in e:
        if c not in x:
            return False
    return True
```

Figure 1: An example EiPE question with the high-level description of "Returns whether a given string contains all vowel letters."

Assume that the variable `x` is a list of `ints`.

```
def f7(x):
    k = x[0]
    x[0] = x[1]
    x[1] = x[2]
    x[2] = x[3]
    x[3] = k
    print(x)
```

Figure 2: An example EiPE question with the high-level description of "Moves the first element of an input list to the end."

to initially focus on specific, functional subsets of the code, akin to expert comprehension strategies like the bottom-up approach. The modified role of variables intervention, in particular, appeared to promote a more effective chunking strategy, which we will detail in Section 4.2.2.

4.1 Students Could Identify Beacons but it Rarely Helped Them

When responding to the beacon intervention prompt, where they were asked to identify and describe the most informative line of code in the program, students identified the same line of code the interviewer believed was the most important line. However, while the students could explain what these lines did in isolation, they struggled to comprehend the relationship between the identified line and other lines in the code.

For instance, participant 5 selected a filtering conditional line within a loop-count plan (see Figure 1, line 4) as the most important line, a line that could potentially give insight into the program's purpose.

> **Participant 5:** (provides an incorrect description) Return True if the vowel number is presented in the input, otherwise return False
> **Interviewer**: Which line of code do you think is the most important?
> **Participant 5**: Uh, if C not in X?

However, this participant could only provide an explanation in the context of the immediate subsequent line.

> **Interviewer**: What do you think that line of code is doing?
> **Participant 5**: So, x is the given string and c is part of this another string. If its not in there. If- So C is like a vowel character. If the character is not in the given string, then return false.
> **Participant 5**: return false if it's not in the input otherwise return true if this, (pointing at c in for c in e line) If C in E, if X not in C.
> **Interviewer**: So what are you thinking right now?
> **Participant 5**: So it tells that for each vowel character, if that character is in the inputs or not.

This participant was not able to reach a high-level understanding of this piece of code, even as they attempted to relate that line to the following line. In this example, as we observed generally, the beacons intervention does not assist students in relating the beacon to the rest of the code.

4.2 Role of Variable Intervention: Inter-connected, Helpful Subsets

The Role of Variables intervention ended up proving effective in some cases. Interestingly, prompting students to identify the roles of variables using their given names did not seem to contribute to their understanding of the code, even if they correctly identified the name. Modifying the intervention to prompt students to explain "the purpose of" each variable in the program helped students focus on useful subsets of the code at a time.

4.2.1 The name of the role is unhelpful. Identifying the name of the variable role did not help them understand the variable in the context of the specific problem, as students could identify the name based on familiar text-surface features without making an attempt to relate it to the code's execution behavior. In the following excerpt, a student expresses exactly the sentiment that we observed while solving the problem in Figure 2, that recalling the role name typically did not help.

> **Participant 22:** (provides an incorrect description) Shifts a four element list over one place to left
> **Participant 22:** (continues) For the use of the [intervention] method, I don't think it really helps for this problem. I guess knowing that k— is it temporary holder? is the only thing that I can see here. It's a temporary variable and that's the only use I can see of it. ... So I would say it's a temporary variable [but] I'm not sure how that would help me in my [explanation]. Like I don't know how [to] include that information in my explanation

Identifying the name does not get students to explain the variable's purpose in the specific problem's context. When we prompt students to explain the purpose of the variable, participants respond with a correct purpose that brings them closer to understanding the purpose of the code as a whole, as in this excerpt from the same interview as above.

> **Interviewer:** Describe what the temporary variable does in this case.

> **Participant 22**: Oh so it holds the first one. Uh, wait, it just— it makes the first element of the list the last one. Yeah. So that that would be the explanation.
> **Participant 22**: (provides an acceptable description of the code) Takes the first element of a list and makes it the last element

Students expressed frustration at recalling the names of the roles that we taught them and how it did not help them understand the problem.

> **Participant 14**: Honestly, it's just too much more to think about. It was annoying to have to recall back to the actual name. I wasn't a fan of the strict names and the definitions, I guess. I don't think it really mattered. My understanding of what the variable did was not going to change based on the name of what the variable is. So it's, like, the understanding of what the variable does is helpful, but it's just the naming convention is too much extra work. And you're going to know what— like, what the variable does, whether or not you call it a fixed variable.

4.2.2 Dividing program into helpful subsets by variable. Prompting participants to explain the 'purpose' of each variable before the entire program helped students comprehend the program. They expressed that it was easier to understand each variable individually before trying to understand the entire program.

> **Participant 18**: I think the purpose of having the variables is so you can differentiate what aspects of the question you're supposed to look at, and then you can kind of use the variables to break up the question in different segments and think about each part. So you kind of just see all the variables, see all the individual parts and break it down and then go through it. And see the outcome.
>
> **Participant 9**: I like the idea of like focusing on what the variables mean. Because it gives some grounding in like how to like approach the problem rather than just like throwing you into some code and like trying to like work through it. It's nice to start somewhere, like, you can to tell students to start with the variables and, like, what do the variables mean and how we defined them and then from there, like, how do we affect the variables.

Prompting students to explain the purpose of a variable resulted in them focusing primarily on lines containing operations involving the variable, thereby identifying a functional, coherent subset of the program. These pieces were sufficiently smaller than the whole, that the student was more likely to be successful in comprehending them even when they could not comprehend the whole program at once. Because these subsets had a coherent purpose, we suspect that students could effectively chunk them after they comprehended them, reducing cognitive load when they later attempted to comprehend the code as a whole.

For example, **participant 33** failed to trace the entire code (Figure 3).

Assume that the variable `x` is a list of `strings`.

```
def Rf12(x):
    o = 0
    g = 0
    for i, w in enumerate(x):
        h = 0
        for c in "aeiou":
            if c in w:
                h += 1
        if h > g:
            g = h
            o = i
    return o
```

Figure 3: An example EiPE question with the high-level description of "Returns the index of the string with the greatest amount of distinct vowels."

> **Participant 33**: Okay, I'm going to mentally use an example, "apple" ... (provides an incorrect description) is it returning the index of, like, the last vowel in a word?

The interviewer prompted them to explain the purpose of the variable h, and they correctly explained it (highlighted in Figure 3).

> **Interviewer**: Can you explain to me the purpose of the variable h?
> **Participant 33**: *h* is the counter of how many vowels ... how many of each type. Like if 'a' appears, we're going to add one. It doesn't matter how many times 'a' appears and it's just going to add one. **how many different vowels are in the word**?

The interviewer then prompted them to explain variable g, which is dependent on the variable h. They successfully applied their understanding of h to help them explain the variable g.

> **Interviewer**: Now explain to me the purpose of the variable g.
> **Participant 33**: (applying understanding of h) Okay, and so if we run on, if *h* is greater than *g*, so right now *g* is at zero. And if we're using **"apple", ... because it has two different vowels in it**, ... if two is greater than *g* which is zero, *g* is now two ... I'm ignoring [variable] o because, I'm not really sure yet- we do it again with the word **"sit" i is the only vowel that appears once and then *h* is in this loop is going to be one** and one is not greater than *g* so *g* is going to stay two because
> **Participant 33:** (gives an acceptable description of *g*) **it's the highest number of different vowels given a word inside of *x***

Understanding the variable g was facilitated by already understanding the variable h, which it depended on.

As another example, **participant 15** initially misunderstood the function (Figure 3) when attempting to comprehend the entire code at once.

> **Participant 15**: (Provides an incorrect description) Returns the index of the last string of the order in a list because it checks for vowels inside each string. Then the end result is *o*, which is just the index of the string that they're currently checking.

The interviewer prompts the participant to explain the purpose of each variable, and the participant performs an abstract trace focusing on the operations performed on the variable h.

> **Interviewer**: Could you explain to me what the purpose of every single variable in this function is?
> **Participant 15**: ... so I realized that for *c* in AEIOU, if *c* is in *w*, so basically *w* in this case, is still the same string.
> **Participant 15:** (comprehending h) So basically **it counted how many distinct vowels were in this string, and** *h* **was the counter of that**.

They comprehend that h is a counter and apply this understanding to comprehend the variable g, which is dependent on the variable h. Finally, they apply their understanding of g to comprehend o.

> **Participant 15**: If *h* is greater than *g*, which initially *g* is zero, so if there's any vowels, this would be the new assigned string. A new assigned value, I should say. *o* was tracking the index of that number. So it basically went through every—
> **Participant 15:** (restating purpose of h) every string in this list of strings checked **how many vowels it was**, and
> **Participant 15:** (comprehended g and o) every time that there was **a string with more vowels than the previous record holder, it would record the index of that string**.

4.2.3 Longitudinal: students no longer need prompting later in the semester. Earlier in the semester (Oct), this longitudinal participant struggled to understand the entire code (Figure 1) without considering subsets first.

> **Participant 4:** (provides an incorrect description) Determines whether there is a vowel in a given string. We're using our for loop to kind of accumulate to test all of the vowels in the string. Testing out the vowels in the string here. And if it's not? If the vowel is not, if any of the vowels are not in *x*, it's false. So True means that there is at least a vowel in *x*.

After the interviewer prompted them to explain each variable, they solved the problem correctly.

> **Interviewer:** Could you explain to me the role of the variable *c*?
> **Participant 4:** It's probably the most recent holder. It's like holding the value a single vowel, so— ohh its a stepper. Interesting. Or. That's looking. OK. So I guess it's a stepper. And then it's just at each for loop [iteration], it's just taking a new step in the vowel progression.
> **Interviewer:** Do you think that might change your explanation?
> **Participant 4:** *c* now will be. So if a vowel. So if "a" return False. Is it gonna? But if it returns false, it won't go back to the return. It won't go back to the loop. Oh, is it? If there is every vowel. Yes. Yeah. That makes more sense.
> **Participant 4:** (provides a correct description) Determines whether there is every vowel in a given string

Later in the semester, the participant would independently explain the purpose of each variable, finding and comprehending meaningful subsets within the program (Figure 3) without needing any prompting from the interviewer.

> **Participant 4:** (comprehended h) *h* is acting as a counting variable. And but that's going to go through each of the vowels, so it's going to **count how many of the vowels are in a word**. Okay, so that's what the for loop is going to do.
> **Participant 4:** (comprehended g) And then with that, if *h* is greater than *g*— it'll define *g* is 0. If *h* is greater than *g*, $g = h$. And then *o* equals *i*. OK, so. It looks like— And then, *o* will hold the place of the index value for... the word, so it counts how many of the vowels are in one of the strings and then... And then, **whichever string has the most vowels, it holds—**
> **Participant 4:** (comprehended o) With the number of vowels— **it holds the number of vowels in that string and the index place for**.

However, they sometimes identified the name of the role of variables incorrectly, further showing that recalling the name of variable roles were sometimes unhelpful and unnecessary.

> **Participant 4:** *g* would be a most recent holder. (should be "most wanted holder") Because we're— that's what we're going to be comparing to.

5 RESULTS: COMPREHENSION VIA CONCRETE TRACING VS MORE ABSTRACT METHODS

Our concrete and abstract tracing interventions helped students make sense of code, but in different ways. Concrete tracing was primarily useful for correcting mistakes students make from reading the code's text-surface features alone. Abstract tracing was useful for identifying relationships between parts of the code and raising an algorithmic understanding of a piece of code to a purpose-level understanding of the code.

5.1 Tracing Maps Text-Surface to Execution

Some students initially misunderstood a program's purpose when only reading its text surface. For example, some students made mistakes when attempting to recognize common patterns based on text-surface. Others made careless mistakes relating to the execution behavior of the code. In a number of these cases,

```
def f1():
    c = 0
    x = 0
    z = 1
    while z != 0:
        z = int(input("Give me a number: "))
        x += z
        if z != 0:
            c += 1
    return x / c
```

Figure 4: An example EiPE question with the high-level description of "Returns the average of all input numbers."

performing a concrete trace of the code enabled them to correct their understanding.

For example, **participant 16** read the code's text-surface and, as a result, falsely assumed the code exits the while loop after the first iteration (Figure 4).

> **Participant 16**: (gives an incorrect description) So then it basically divides the number represented by x, which is essentially zero added to the number that was in the input, and then it divides that input by one, which is just going to be the input. So it asks the user for an input and then essentially returns that number.

Performing a concrete trace allowed them to identify the mistake, which enabled them to progress to a correct understanding of the code.

> **Interviewer**: So could you try inputting like a bunch of numbers for z?
> **Participant 16**: (performs concrete trace) So for example, if you have like 4, for example, as z, then while z, or so then x plus or equals z. Then x becomes 4. And then if z doesn't equal 0, which it doesn't, then c plus or equals 1. So then c becomes 1. And then, for example, if you have like 6 ...
> **Participant 16:** (explains algorithm) Oh, so then it would go— So it essentially, like, goes one by one— like, based on how many times the function runs it— like, divides the new x by that number and then returns it. After I put in a series of numbers and my output was— I realized that the outputted numbers were coming from essentially, like, a total that was divided by the count of how many times the function had run and then the total was divided by the count
> **Interviewer**: Do you think you have an even, like, shorter way of explaining that to me now?
> **Participant 16**: (gives acceptable answer) **returns a total divided by the number of times a function has run**.

While we have misgivings about the student's use of vocabulary in their final answer, it is clear they are referencing the computation of an average.

When prompted to trace, participants commonly realize their mistake in misrecognizing patterns. For example, when participant 20 traced the "count distinct occurrences" pattern (the purpose of variable h, Figure 3), they realized it did not count duplicate occurrences, correcting their misunderstanding.

> **Participant 20**: (gives an incorrect description) so I think it's counting the amount of vowels in each word in list x. And I'm pretty sure it's returning the word, the index of the word that **has the most vowels**.
> **Interviewer**: Can you explain to me the variable h again, please?
> **Participant 20**: (gives an incorrect description) I'm pretty sure h is like a counter for the vowels. Well, for the, for the, well, how many vowels are in w, which is the word, which is the string in this x. I think that's what h is. Kind of like a, you know, counter. It counts for each word. No, for each string in this x, **it counts how many vowels there are in there.** I'm guessing.
> **Interviewer:** (prompts to trace)
> **Participant 20**: (traces line-by-line) ... Then h equals 3. Oh wait, there's... it's coming twice. There's four. I'm so confused. Ah, is it? Let's say c is 'a' now. 'a' appears. h is 1. Then we have 'e'. That appears. We have 2. Then we have 'i'. That doesn't appear. We have 'o'. That appears twice. Does that mean we just count? it once or twice? Does it count the word that has all the vowels in there? No, this is. Yeah, so I'm thinking ... it counts the word has, ***not* the most vowels, like number wise, but like, it has most vowels *singularly***. And then return the index of that.

We observed multiple instances (including the above) where students misrecognized patterns, which could be attributed to students aggressively pattern matching to the patterns taught in the course (e.g., a simple counting pattern).

> **Participant 42**: So I think it's just the fact that, like, from last semester, I was just accustomed to the fact that these type of functions where it's counting, it's looking for the total amount of something rather than, like, the unique amount.

5.2 Tracing Pitfall: Students Choose a Limited Set of Inputs Confirming False Hypotheses

Some students attempted to understand code through inductive reasoning, by looking for patterns in multiple input-output pairs. Students who successfully used inductive reasoning chose a diverse set of inputs that exposed the full functionality of the code. In contrast, unsuccessful students seemed to make false assumptions based on their read of the code's text-surface (syntax) or their first few inputs, then chose a limited set of inputs to confirm their false assumptions. To help students choose appropriate inputs, the interviewer asked them to find additional inputs that could contradict their current (incorrect) belief of what the code did, which seemed to encourage them to choose more diverse inputs.

Assume that the variable x is an int.

```
def f19(x):
    o = 0
    while x > 0:
        if (x % 10) % 2 == 0:
            o += 1
        x //= 10
    return o
```

Figure 5: An example EiPE question with the high-level description of "Returns the number of even digits in a given input integer."

5.2.1 Successful students choose additional inputs of new characteristics to contradict their explanation. For example, participant 39 incorrectly explained the code in Figure 5 after appearing to read its syntax and chose a limited set of inputs consisting of mostly 0 digits, confirming their incorrect explanation repeatedly.

> **Participant 39**: I don't know the full purpose, but ... (reads code verbatim) returning how many times the iteration or each division makes the number even maybe?
> **Interviewer**: Would you like to try an input?
>
> **Participant 39**: (tries input 100) I still have the same idea that I did in the beginning.
> **Interviewer**: Would you like to try another input?
> **Participant 39**: (tries input 1000) I still have the same idea as before. It hasn't changed.

Upon prompting from the interviewer, the participant chose a more diverse set of inputs that now contained odd numbers and then understood the code correctly, refuting their initial incorrect explanation.

> **Interviewer**: So now can you try out an input that might make you change your explanation to make it better. Like, think of an input. that could help you better to change your explanation.
> **Participant 39**: Yeah, I could probably **choose like an odd number** or something.
> **Participant 39**: (**tries input 55**) Odd numbers so we don't increment. For some reason now I'm thinking it has something to do with digits. It was 55 and then five. I think it has something to do with digits just because like what I've seen with the even inputs in this, but— **Could it be that we're checking if each digit is even?** So every time a digit is even, we're adding one to *o*. The digits, like, 55 over 10 would give us like 5. We're checking, we're incrementing *o* every time, each digit from right to left is even.

The prompt from the interviewer to try inputs that could add to their understanding of the problem seemed to get the student to try inputs of more varying characteristics (e.g., odd numbers rather than numbers ending with 0s).

Assume that the variable x is an int.

```
def f4(x):
    for i in range(2,int(x/2)+1):
        if x % i == 0:
            return False
    return True
```

Figure 6: An example EiPE question with the high-level description of "Returns whether the given input is a prime number."

As another example, **participant 32** traced two inputs of odd numbers, falsely assuming the function (Figure 6) would return true for all odd numbers.

> **Participant 32**: (tries inputs 3 and 5) If the number is odd— Would it just return true if the number is odd?

They try even numbers, which confirms their incorrect hypothesis.

> **Participant 32**: (tries inputs 4 and 6) So if we input an even number, Then it's gonna return false for any even number input. If it's odd it's going to return true.

The interviewer prompts them to find an input that could invalidate their explanation. They initially consider trying another even number, hesitate, and then decide to try an odd number (9) instead, leading them to realize their previous explanation was incorrect and that they must try more odd numbers to understand the code.

> **Interviewer**: Now try out another input that you think might disprove this hypothesis that you have.
> **Participant 32**: Um, maybe, okay, if I put in 10, no, an odd number. If I put in 9. then it's going to return false. So it's not going to return true for all odd numbers. So it's going to return true for some odd numbers, but I'm not sure which ones without just like testing all of them. Three is going to return true. Five is going to return true. 7, ... oh, is it? No, wait. ... So that would return.
> **Participant 32**: (gives a correct description) Oh, is it checking if the odd numbers have any multiples? Is it a prime number? Is it returning true if the number is a prime number?

After trying some more odd inputs, the participant then understands the code correctly. Asking students to attempt to disprove their hypotheses about the code's purpose seemed to guide the student to select inputs that helped to resolve their misunderstanding.

5.3 Abstract Tracing Shows the Functional Relationships Between Variables

Some students, who relied on inductive reasoning (i.e., generating input/output pairs without deducing relationships between lines), failed to recognize input-output patterns. The abstract tracing intervention enabled some of these students to understand the functional relationships (intent-dimension) between variables.

It demonstrates how the action performed by each variable collectively contributes to the code's overall functionality, such as how variable dependencies constrain actions and how independent relationships allow actions to complement each other. With this approach, the structural relations (execution-dimension) between variables can be understood not just for specific input-output cases but for any conceivable inputs.

In the excerpt below, the interviewer attempted to correct a participant's misunderstanding of the code in Figure 4. Prompting the student to explain the purpose of each variable and perform concrete tracing was unsuccessful.

> **Interviewer**: So what is your high-level explanation? How do you explain its purpose?
> **Participant 41**: The purpose is to return numbers but to ensure that you don't get any undefined numbers at the end because it does make you return a division of some sort.
> **Interviewer**: So could you explain to me the purpose of the variable z, x, and k?
> **Participant 41**: (responds with a line-by-line explanation)
> **Interviewer:** (prompts to concretely trace)

(pre-loop code)	(first iteration)	(second iteration)
`c = 0`	`c = 1`	`c = 2`
`x = 0`	`x = 2`	`x = 5`
`z = 1`	`z = 2`	`z = 3`

> `2.5` (return value)
> **Participant 41**: I'm trying to figure out if I see some sort of trend, but I really don't

However, after performing the abstract trace intervention, where the sum variable was noted as a list of separate symbols, the participant gained a key insight. They realized that **the total number of integers added would always equal the counter variable**, *regardless of the specific integer inputs.* This demonstrated the functional, independent relationship between the sum and counter variables and consequently revealed the overarching computation of calculating the **average**.

> (**Interviewer** prompts to use abstract tracing intervention)

(pre-)	(first)	(second)	(third)
`c = 0`	`c = 1`	`c = 2`	`c = 3`
`x = 0`	`x = S1`	`x = S1 + S2`	`x = S1 + S2 + S3`
`z = 1`	`z = S1`	`z = S2`	

> **Participant 41**: (gives a correct description) is it an **average** of the values?
> **Participant 41**: because... **S2. And then the c is also two.** And then I tried it with **S3 then made c 3.** And I kind of was able to actually see the trend of it being an average.

Noting un-collapsed expressions demonstrated how the action of taking the sum complemented the action of the counter to make the average. When viewing the total accumulated as one value initially, they found it unclear how the counter variable `c` relates to the summing variable `x`.

Assume that the variable `inputList` is a list of `ints`.
Assume that the variable `k` is an `int`.

```
def f(inputList, k):
    x = 0
    y = 0
    for i in range(len(inputList)):
        x += inputList[i]
        if i >= k-1:
            if x > y:
                y = x
            x -= inputList[i-k+1]
    return y
```

Figure 7: An example EiPE question with the high-level description of "Returns the sum of the largest k consecutive elements."

> When I did it the first time, I actually physically added the numbers together. So I wasn't able to notice it being an average.

As another example, **participant 36** seemed to struggle to understand the code in Figure 7, even after tracing with multiple inputs.

> **Participant 36**: (performs concrete tracing with inputList as `[4,3,5,7,1]` and `K` as 1 and 2, not shown)
> **Participant 36**: (gives an incorrect description) Return the maximum integer in the list + the sum of each integer in the list positioned before a user provided index.
> **Participant 36**: (traces with inputList `[4,3,5,7,1]`, and `K` as 3, shown below, read left-to-right)

`i = 0`	`i = 0`	`i = 1`	`i = 2`	`i = 2`	`i = 2`
`x = 0`	`x = 4`	`x = 7`	`x = 12`	`x = 12`	`x = 8`
`y = 0`	`y = 0`	`y = 0`	`y = 0`	`y = 12`	`y = 12`

`i = 3`	`i = 3`	`i = 4`	`i = 4`
`X = 15`	`X = 12`	`X = 13`	`X = 8`
`Y = 15`	`Y = 15`	`Y = 15`	`Y = 15`

> **Participant 36**: It's still hard to see what exactly it's doing. Because the way my brain is processing this is more like a math problem. Where it's like you do a series of steps and you get an answer. How do you get from A to B without describing every single step in between.

When they perform the abstract tracing intervention, they begin to understand the action performed by the variable `x` (**move forward/right, check every 3 numbers**) and its functional, dependent relationship with the max variable `y` (**all add to the highest value**).

> **Interviewer:** (suggests to use intervention, uses concrete numbers but does not simplify expressions)
> `inputList = [4,3,5,7,1]`

```
K = 2
X = 4 + 3        X = 3 + 5        X = 5 + 7
Y = 7            Y = 8            Y = 12
```

Participant 36: X starts at 7 and then well **this would move right**. So 8. X would continuously just update to equal **the sum of every pair.**

```
inputList = [4,3,5,7,1]
K = 3
X = 4 + 3 + 5    X = 3 + 5 + 7    X = 5 + 7 + 1
Y = 12           Y = 15           Y = 15
```

Participant 36: **You go forward. And then you go back. It checks every three numbers. the surrounding numbers all add to the highest value** (3 + 5 + 7 = 15).

Displaying the sum as un-collapsed expressions demonstrated the action of x (move forward, back, every 3 numbers) and its functional, dependent relationship to the max variable y (surrounding three numbers 3 + 5 + 7 all add to the highest value). They continue the same strategy now with K = 4 to understand the functional, dependent relationship between all possible input cases of variable k (**'if I change k, what changes?'**), x, and y.

Participant 36: So what is it actually returning? What does K have to do with it? Because **if I change K, what changes?** If I change K to 4, the same pattern would hold true, only it wouldn't start subtracting until here. Because it's the fourth value.

```
inputList = [4,3,5,7,1]
K = 4
X = 4 + 3 + 5 + 7        X = 3 + 5 + 7 + 1
Y = 19                   Y = 19
```

Participant 36: **So would y return the highest set of numbers in a row equal to the length of k.**
Interviewer: So how did you manage to reach your explanation?
Participant 36: ... if it's adding a value, then **removing it**, that **it's like the values that track in**, just moves an index over each time. And so once I saw that, I was like, oh, okay, so it's adding three numbers and **Y is incrementally— is just going to increase if every three would be higher**. And then, so after figuring out what, Y was actually like being, I guess **after noticing that pattern of like threes**, I connected it. I was like, okay, what, **where does K has something to do with that? So then K is three. I then tested it with four**.

They mention the general action of variable x, which is how it acts as a "track" that adds and removes (instead of "subtracts") values. They explain the functional relationship between the "track" sum, the max variable y (highest set of numbers in the "track"), and input k ("track" length of k). Un-collapsed expressions demonstrate how k constrains x ("track" length k is the number of added items x) and how the max y depends on (is constrained by) the largest possible sum x of k items.

5.3.1 Longitudinal: students independently use intervention, no need for prompting. At earlier points during the semester, participant 13 initially misunderstood some variables in the code in Figure 3. After the interviewer prompted them to use the intervention, they corrected their understanding, correctly explaining the functional, dependent relationship between the variables h and g in every input case.

Participant 13: G is the greatest number of vowels. Returns the index of the one with the greatest number of vowels
Interviewer: (prompts to use strategy)

```
o = 0
g = 0
h = 0
C = Vx "aeiou"
H = 1-5
G = 1-5
O = index
```

Participant 13: So I guess it's not the greatest number of vowels total. No varying. No. Different types of vowels. Since I had with the most number of vowels period, that number could exceed 5. So when I had it written out, I saw that it has to be between one and five. So it's not number, it's variety.

Participant 13's behavior changed in the later part of the semester, as they started using the intervention independently when they seemed to lack confidence, in contrast to earlier instances when they displayed overconfidence and required prompting from the interviewer to use the intervention.

Participant 13: I'm going to use the strategy because I'm stuck

Additionally, participant 13 demonstrated the ability to mentally perform abstract traces and comprehend variable relationships without note-taking in their solution to the code in Figure 8. This may demonstrate the participant has advanced as a programmer, capable of "chunking" or fitting more items in their working memory per the cognitive load theory [48].

Participant 13: If string 1 count of the letter i is lower. x plus string count i. Why would they do the lower more? OK, I think I get it. It's the amount of letters that the strings have in common. When I realized that it was the lowest count. And since we're returning, it was easier because I didn't have to focus on all the other parts of the code. I was mainly focusing on X and I could see that like X was an integer, so it's counting something. So because they said it as zero, so as a counter. And then they were counting the smaller one … the smaller one has the max amount of similar letters. If you did the greater one, that wasn't all the ones they have in common, but if you go with the smaller ones, it's all the letters that they have in common.

Assume that the variables s1 and s2 are strings.

```
def f(s1, s2):
    x = 0
    y = []
    for i in s1:
        for j in s2:
            if i == j and i not in y:
                if s1.count(i) < s2.count(i):
                    x += s1.count(i)
                else:
                    x += s2.count(i)
                y.append(i)
    return x
```

Figure 8: An example EiPE question with the high-level description of "Returns the number of common characters between two strings."

5.4 Algorithmic Visualizations Show Variable Actions & their Functional Relationships

We observed cases where participants correctly traced or abstractly traced but could not explain the program's purpose. When the participant performed a note-taking technique resembling an algorithmic visualization (e.g., highlighting, pointing, and gesturing on inputs), they successfully explained the purpose of the code. Their drawn visualization demonstrated the *action* performed by each (role of) variable using *gesture-like* techniques. Similar to the abstract tracing intervention, it also demonstrates how their functional relationships make up the program's intent.

For example, participant 29 correctly traced the code (Figure 8) with inputs but could not understand the purpose.

> **Interviewer:** (suggests to concretely trace)
> **Participant 29**: (tries "123456789", "133444564", "1234456789", " 133444564")
> **Participant 29**: This is stupid, right? If I did that, that would be six. four, five, six. How could this be useful? (tries "Helloworld", "Hellothere")
> **Participant 29**: I'm trying with letters. I'm not seeing any pattern here. I'm trying to find a pattern between the letter and the numbers. Well, I'm not really seeing anything here.

They used a note-taking technique to visualize the algorithm, highlighting common characters between two strings, helping them realize that the sum variable x was counting the common set of characters between two strings.

> (Begins to highlight inputs and gesture)
> "123445678 9" "Helloworld"
> "133444564" "Hellothere"
> 134456 Hello r
>
> **Participant 29**: So it's whatever they have in common. Returns the amount of same letters or numbers in the string.

The variable y prevents duplicate characters from being counted, which constrains the action done by x to count only the common characters. The drawn visualization demonstrates this constraint by showing how some occurrences of the numbers 3 and 4 are not highlighted in the first string. Thus, highlighting demonstrated the overarching action of counting the common set of characters based on the functional relationship between the sum variable x and the gatherer variable y.

As another example, **participant 9** incorrectly explains the function (Figure 7), not taking into account the "most wanted holder" (max) variable y correctly (highlighted in Figure 7).

> (traces below)
> K = 2
> [1, 2, 3, -3, 4]
> **Participant 9**: So its the sum of the last positive K terms.

Afterward, they use a note-taking strategy that resembles an algorithmic visualization. This technique demonstrates the functional relationship between the most wanted (max) holder y and the sum variable x, showing the current set of k contiguous elements checked against the current maximum sum. The visualization technique highlights when the most wanted holder does not update due to a negative number, demonstrating the constraint of when the maximum value (find the best) does not update. The participant then correctly integrates their understanding of the find the best (maximum) variable to their explanation.

> (Interviewer asks to try again)
> K = 2
> [1, 2, 3, -3, 4]
> [1, 2, 3, -3, 4]
> [1, 2, 3, -3, 4]
> **Participant 9**: So I guess that does change how I think of it. It would still be five,
> [1, 2, 3, -3, 4]
> **Participant 9**: But yet we're all the way here. What does that mean? *[pause]* So, like, essentially it's gonna find the largest, like, streak of a number of numbers. This function finds the largest sum of k number of elements in succession.

6 DISCUSSION / CONCLUSION

Based on our analysis of the interviews, we make the following suggestions as to how we might go about teaching novices to comprehend code. The following should be considered as no more than conjecture and will need to be validated through experimental or quasi-experimental study.

6.1 Teach Students a Collection of Tools

Students failed to comprehend code for a wide variety of reasons ranging from incorrectly recognizing patterns to failing to abstract relationships between lines of code. It is our belief from our observations that these different kinds of failures have different root causes and that no single intervention can address all of them. As such, it will be useful to our students to teach them a collection of

techniques so that whatever the issue, they will have a technique that can address it.

6.2 Teach Students to Analyze the Code One Variable at a Time

Our participants often attempted to comprehend the entire program at once, reading code linearly top-to-bottom, left-to-right, similar to novices in prior eye-tracking studies [14], and frequently struggled to do so, by what appeared to be cognitive overload. They would make careless mistakes tracing and misrecognizing patterns. Since novices lack extensive plan knowledge, they do not automatically 'chunk' code into useful sub-goals like experts [48].

Prompting students to explain the purpose of each variable provided them a straightforward way to focus on a useful subset of the code at a time, with each subset appearing to represent a functional, purposeful sub-goal [44] of the program. We did not have to teach them how to perform program slicing [62] by variable, they just seemed to do it naturally, at least for these small programs. Our participants used concrete or abstract tracing on these subsets to comprehend them and effectively used the identified purpose of one variable in the comprehension of dependent variables. Often, when directed to "explain the purpose of every single variable in this function," our students visited the variables in a useful order (i.e., dependent variables after the variables they depend on), but we have not explored this enough to understand if this is an ability that also comes naturally to novices or if some aspect of our code fragments facilitated this result.

Although variables may not universally correspond to sub-goals, we found thinking of this strategy as a way of extracting the sub-goals of a piece of code and then naming them, as a useful metaphor. Further study to reconcile existing findings related to sub-goals [27, 28] and these findings is likely profitable.

That identifying the names of the role of variables rarely helped students understand the purpose of code is consistent with the Block Model [18]. Naming the role of a variable is an activity at the structural level of the program, which is one level removed from the purpose of the program. In contrast, asking students the purpose of the variable in the context of a given program is in the correct dimension, just smaller in granularity than the program as a whole.

6.3 Teach Students to Try to Disprove Their Hypothesis of What the Code Does

Participants who read syntax (text-surface) without tracing were more likely to make false assumptions about execution behavior. Novices make false assumptions and stick to their incorrect mental models [11, 36, 59], unlike experts. Thus, caution is necessary for novices when attempting to recognize code based on its text surface. Novices should be encouraged to perform a concrete trace of code to check their hypothesis. (We assume that most introductory programming courses teach tracing as a means of internalizing a language's semantics.)

We observed intriguing transformations in student behavior through our longitudinal studies. At earlier points in the semester, one longitudinal participant (13) would appear to display overconfidence, immediately answering the problem incorrectly and needing prompting to use the intervention. Later in the semester, they seemed to display caution and would independently decide to use the intervention. We speculate that their self-awareness of their lack of understanding and that they needed to switch strategies (use the intervention) is a sign of "monitoring your own understanding," one of the signs of metacognition [7]. We propose that such caution could be beneficial for novices as it may encourage rigorous verification of understanding and a readiness to refute questionable hypotheses [11, 36, 59]. This caution should specifically target their intermediary hypotheses of the problem, not their belief in their abilities or confidence in eventually solving the problem correctly.

Some students rely on inductive reasoning (concrete tracing of multiple inputs) to understand code. When tracing to understand code, students must choose a diverse set of inputs to expose the general behavior of the code [15]. Like prior work [15], we found that students may choose a narrow set of inputs to confirm their false assumptions, demonstrating confirmation bias [34]. We found a potential solution to such a problem: prompting them to "choose an input to prove yourself wrong" seems to encourage them to select more diverse inputs, refuting their incorrect assumptions and correcting their understanding of the problem. This finding is in-line with the literature on falsification-driven verification and unit testing [13], which follows Popper's theory that we must seek to repeatedly falsify claims and that science must be grounded in a constant search for counterexamples [35]. We should teach students to be skeptical of their hypotheses, testing multiple inputs to refute faulty intermediary hypotheses.

Besides confirmation bias, another issue students had with inductive reasoning was not recognizing input-output patterns. We found that abstract tracing and algorithmic visualizations, which helped them understand relationships between variables, may serve as a potential solution. Additionally, when we prompted them to explain "the purpose of each variable in this code," we did not have to explicitly teach our students the concepts of code coverage and boundary conditions to have them select a more diverse set of inputs. This framing alone seemed to help them identify more diverse and profitable inputs.

6.4 Teach Students to Perform Abstract Tracing

Some participants do not understand the purpose of a piece of code despite tracing it correctly, like the "neglected middle novice" [22]. While the "neglected middle novice" can trace correctly, they struggle to abstract meaningful relationships between lines of code, solely relying on input-output inference [54–56]. Bennedsen et al. [2] observed no significant differences in novices' code tracing performance using a debugger versus manual tracing, speculating that the debugger is only useful for identifying execution errors and not for understanding object interaction. Our results are consistent with their findings and suggest the need for more abstract strategies, such as abstract tracing or algorithmic visualizations, to comprehend the relationship between variables / lines of code.

As such, we suggest that students be taught to trace symbolically. In our observations, a key feature of successful uses of abstract tracing was not collapasing / simplifying arithmetic expressions. Simplified expressions can obfuscate the evolution of variables and their relationships in ways that the unsimplified expressions do not.

Abstract tracing can be employed when a student has confirmed their understanding of the behavior of a piece of code (i.e., the structure dimension in the Block model) but is struggling to identify its purpose.

Prior research suggests that programmers develop the skill to symbolically reason about code as they gain a deeper understanding of semantics [50, 55]. We found our participants incorrectly explaining the semantics of a program initially, then tracing, then correctly abstract tracing to correct their understanding of the purpose of the program. This suggests that our participants had a correct understanding of the operational semantics of our programs if tracing and abstractly tracing resolved their misunderstanding, demonstrating that students fail to mentally abstractly trace or recognize code primarily due to their false assumptions they make based on reading the text-surface. We could potentially teach students to practice abstract tracing with no need for a tool [8] by first tracing code with multiple inputs, then after developing a sufficient understanding of the code, abstractly trace the same code by tracing using symbols. This also further supports that students should consider code execution behavior until they develop sufficient plan and semantic knowledge.

In some cases, students required more abstract strategies to discern and comprehend the functional relationships between key actions within the code. This observation aligns with previous studies, which suggested that experts know when and how to reason about code at the appropriate level of abstraction for a given problem [58], focusing on key actions, their structural relationships, and constraints [38]. When the level of abstraction is too low for both the problem and the learner, the learner may become excessively fixated on specific, irrelevant surface features, thereby obscuring the underlying structural relationships. In our study, certain participants who engaged in concrete tracing may have been overly occupied with computing mathematical operations on only one input case, thus preventing them from recognizing broader, key relationships within the code. The abstract tracing intervention, which used un-collapsed expressions and symbols representing broader ranges of inputs, enabled them to focus on higher-level actions done by each variable and their relationships.

Analogy recognition [38], a cognitive process integral to learning and knowledge transfer, involves discerning common structural patterns and relations between distinct problems or contexts. Prior research on cognition found that having a more abstract understanding of problems promotes analogical transfer [26, 43, 45]. We speculate that more abstract strategies (abstract tracing and visualization) promoted such a type of recognition of concepts. Additionally, prior work found that demonstrating mathematical concepts through actions and gestures promoted learning [12], consistent with our findings of students performing gesture-like actions in their drawn visualization to help them recognize key actions done by variables and their relations within their concrete trace.

While we saw instances of students using algorithmic visualizations to understand the purpose of code, we are not ready to recommend that we attempt to teach students this approach. Previous research [41, 50] has found that animations illustrating variable roles can improve students' understanding, and allowing them to control the animations can enhance their mental model of the notional machine. Our key concern is that we have not yet identified a structured process students could undertake to reliably produce useful visualizations. Suggesting students "draw something to help them understand the code" is likely too vague to be useful for novices [50, 58]. Furthermore, novices may struggle with finding the appropriate level of abstraction to solve problems [51]. When our participants successfully used abstract visualization methods, they did so after tracing the program accurately, indicating that tracing may serve as a stepping stone to visualizing abstract representations. We think that this is exciting area for future study, but we do not yet have suggestions of how to actualize our findings in this area.

ACKNOWLEDGMENTS

Thanks to Vivian Van der Werf, Violetta Lonati, Carl-Haynes Magyar, Kristin Stephens-Martinez, Jean Salac, Patrick Wang, William Kunkel, Diana Franklin, Barbara Ericson, and various members of the ICER community for giving very helpful feedback during Mohammed Hassan's doctoral consortium. This material is based upon work supported by the National Science Foundation under Grant No. DUE 21-21424 and Mohammed Hassan's SURGE and graduate college fellowships at the University of Illinois.

REFERENCES

[1] Mordechai Ben-Ari and Jorma Sajaniemi. 2004. Roles of variables as seen by CS educators. *ACM Sigcse Bulletin* 36, 3 (2004), 52–56.
[2] Jens Bennedsen and Carsten Schulte. 2010. BlueJ visual debugger for learning the execution of object-oriented programs? *ACM Transactions on Computing Education (TOCE)* 10, 2 (2010), 1–22.
[3] Sarah Blankenship. 2022. Learning to Reason About Code with Assertions: An Exploration with Two Student Populations. (2022).
[4] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.
[5] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265.
[6] Pauli Byckling and Jorma Sajaniemi. 2006. Roles of variables and programming skills improvement. *ACM SIGCSE Bulletin* 38, 1 (2006), 413–417.
[7] John H Flavell. 1979. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American psychologist* 34, 10 (1979), 906.
[8] Megan Fowler, Jason Hallstrom, Joseph Hollingsworth, Eileen Kraemer, Murali Sitaraman, Yu-Shan Sun, Jiadi Wang, and Gloria Washington. 2021. Tool-Aided Learning of Code Reasoning with Abstraction in the CS Curriculum. *Informatics in Education* 20, 4 (2021).
[9] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* (2022), 1–29.
[10] Edward M Gellenbeck and Curtis R Cook. 1991. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*. Ablex Publishing, Norwood, NJ, 65–81.
[11] David Ginat. 2001. Misleading intuition in algorithmic problem solving. *ACM SIGCSE Bulletin* 33, 1 (2001), 21–25.
[12] Susan Goldin-Meadow. 1999. The role of gesture in communication and thinking. *Trends in cognitive sciences* 3, 11 (1999), 419–429.
[13] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. 2018. How verified (or tested) is my code? Falsification-driven verification and testing. *Automated Software Engineering* 25 (2018), 917–960.
[14] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 171–174.
[15] Mohammed Hassan and Craig Zilles. 2023. On Students' Usage of Tracing for Understanding Code. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 129–136.
[16] Vighnesh Iyer and Craig Zilles. 2021. Pattern Census: A Characterization of Pattern Usage in Early Programming Courses. In *Proceedings of the SIGCSE Technical Symposium (SIGCSE)*.
[17] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. 2017. On the ability to reason about program behaviour: A think-aloud study. In *Proceedings of the 2017 ACM*

Conference on Innovation and Technology in Computer Science Education. 305–310.

[18] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 2019. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 27–52.

[19] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education (TOCE)* 22, 2 (2021), 1–33.

[20] Marja Kuittinen and Jorma Sajaniemi. 2004. Teaching roles of variables in elementary programming courses. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. 57–61.

[21] S Letovsky. 1986. Cognitive Processes in Program Comprehension: First Workshop. E. Soloway and S. Iyengar eds.

[22] RF Lister. 2007. The neglected middle novice programmer: Reading and writing without abstracting. *National Advisory Committee on Computing Qualifications* (2007).

[23] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. ACM, New York, NY, USA, 161–165. https://doi.org/10.1145/1562877.1562930

[24] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental models and software maintenance. *Journal of Systems and Software* 7, 4 (1987), 341–355.

[25] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ACM, 101–112.

[26] Jean M Mandler and Felice Orlich. 1993. Analogical transfer: The roles of schema abstraction and awareness. *Bulletin of the Psychonomic Society* 31, 5 (1993), 485–487.

[27] Lauren Margulieux and Richard Catrambone. 2017. Using learners' self-explanations of subgoals to guide initial problem solving in app inventor. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 21–29.

[28] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* 7 (2020), 1–16.

[29] Michael McCracken et al. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Canterbury, UK) *(ITiCSE-WGR '01)*. ACM, New York, NY, USA, 125–180.

[30] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM conference on international computing education research*. 2–11.

[31] Michael P O'Brien, Jim Buckley, and Teresa M Shaft. 2004. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 6 (2004), 427–447.

[32] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop, 1987*. 100–113.

[33] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

[34] Rüdiger F Pohl. 2022. Cognitive Illusions. *Cognitive Illusions: Intriguing Phenomena in Thinking, Judgment, and Memory* (2022), 3.

[35] Karl Popper. 2005. *The logic of scientific discovery*. Routledge.

[36] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.

[37] Michael Raadt, Mark Toleman, and Richard Watson. 2007. Incorporating programming strategies explicitly into curricula. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*.

[38] Nikolaus Ritt. 2005. Analogy and Transfer: Encoding the Problem at the Right Level of Abstraction. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 27.

[39] Jorma Sajaniemi. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE, 37–39.

[40] Jorma Sajaniemi, Mordechai Ben-Ari, Pauli Byckling, Petri Gerdt, and Yevgeniya Kulikova. 2006. Roles of variables in three programming paradigms. *Computer Science Education* 16, 4 (2006), 261–279.

[41] Jorma Sajaniemi and Marja Kuittinen. 2003. Program animation based on the roles of variables. In *Proceedings of the 2003 ACM symposium on Software visualization*. 7–ff.

[42] Jorma Sajaniemi and Marja Kuittinen. 2005. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education* 15, 1 (2005), 59–82.

[43] Emmanuel Sander and Jean-François Richard. 1997. Analogical transfer as guided by an abstraction process: The case of learning by doing in text editing. *Journal of experimental psychology: learning, memory, and cognition* 23, 6 (1997), 1459.

[44] Carsten Schulte. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research*. 149–160.

[45] Daniel L Schwartz. 1993. The construction and analogical transfer of symbolic visualizations. *Journal of research in science teaching* 30, 10 (1993), 1309–1325.

[46] Nianfeng Shi. 2021. Improving undergraduate novice programmer comprehension through case-based teaching with roles of variables to provide scaffolding. *Information* 12, 10 (2021), 424.

[47] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'souza. 2009. A taxonomic study of novice programming summative assessment. In *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95*. Citeseer, 147–156.

[48] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[49] Elliot Soloway, Beth Adelson, and Kate Ehrlich. 1988. Knowledge and processes in the comprehension of computer programs. *The nature of expertise* (1988), 129–152.

[50] Juha Sorva et al. 2012. *Visual program simulation in introductory programming education*. Aalto University.

[51] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–64.

[52] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[53] James C Spohrer, Elliot Soloway, and Edgar Pope. 1985. A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction* 1, 2 (1985), 163–207.

[54] Donna Teague. 2015. Neo-Piagetian theory and the novice programmer. *Diss. Queensland University of Technology* (2015).

[55] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]*. Australian Computer Society, 87–95.

[56] Donna Teague and Raymond Lister. 2014. Blinded by their Plight: Tracing and the Preoperational Programmer.. In *PPIG*. 8.

[57] Donna Teague and Raymond Lister. 2014. Longitudinal think aloud study of a novice programmer. In *Conferences in Research and Practice in Information Technology Series*.

[58] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin* 39, 3 (2007), 236–240.

[59] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[60] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[61] Renske Weeda, Sjaak Smetsers, and Erik Barendsen. 2023. Unraveling novices' code composition difficulties. *Computer Science Education* (2023), 1–28.

[62] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[63] JL Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. (2011).

[64] Susan Wiedenbeck. 1991. The initial stage of program comprehension. *International Journal of Man-Machine Studies* 35, 4 (1991), 517–540.

[65] Susan Wiedenbeck and Jean Scholtz. 1989. Beacons: A knowledge structure in program comprehension. In *Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.)*. 82–87.

[66] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.