Students' Misunderstanding of the Order of Evaluation in Conjoined Conditions

Eliane S. Wiese
School of Computing
University of Utah
Salt Lake City, USA
eliane.wiese@utah.edu

Anna N. Rafferty
Computer Science Department
Carleton College
Northfield, USA
arafferty@carleton.edu

Garrett Moseke
School of Computing
University of Utah
Salt Lake City, USA
gmoseke@gmail.com

***Abstract*—Experts often use particular control flow structures to make their code easier to read and modify, such as using the logical operator AND to conjoin conditions rather than nesting separate `if` statements. Within Boolean expressions, experts take advantage of short-circuit evaluation by ordering their conditions to avoid errors (such as checking that an index is within the bounds of an array before examining the value at that index). How well do students understand these structures? We investigate students' use and understanding of conjoined versus separate conditions within a larger assessment of 125 undergraduate students at the end of their second- and third-semester CS courses (in algorithms & data structures and introductory software engineering). The assessment asked students to: write code where an edge case error could be avoided with short-circuit evaluation, revise their code with nudges towards expert structure, and answer comprehension questions involving code tracing. When writing, students frequently forgot to check for a key edge case. When that case was included, the check was often separated in its own `if`-statement rather than conjoined with the other conditions. This could indicate a stylistic choice or a belief that the check had to be separated for functionality. Notably, students who included all necessary conditions rarely exhibited the error of ordering them incorrectly. However, with code comprehension, students demonstrated significant misunderstandings about the effects of condition ordering. Students were more accurate on comprehension tasks with nested `if`s than conjoined conditions, and this effect was most pronounced when the ordering of the conditions would lead to errors. When conditions were conjoined in a single expression, only 35% of students recognized that checking a value at an index before checking that the index was in bounds would lead to an error. However, 54% of students recognized the problem when the conditions were separated into individual `if`-statements. This demonstrates a subtlety in code execution that intermediate students may not have mastered and emphasizes the challenges in assessing students' understanding solely via the way they write code.**

***Index Terms*—Computer science education, Novice code comprehension, Programming style**

I. Introduction

Two goals of teaching programming in computer science beyond having students write code that works are for students to understand how code is executed and to be able to read code they did not write. That is, students need an accurate mental model of the *notional machine* – an abstraction of the system that takes code as input and produces a running program [1], [2]. One measure of students' understanding comes from their choices of structures and control flows when they write code [3]. However, code-writing skills alone may not be an accurate measure of students' understanding of code execution. Students may comprehend more advanced code structures even when they don't write with them [4], and, as we show here, students may not fully comprehend the code structures that they use correctly when they write.

General theoretical frameworks in the learning sciences provide a grounding for why code writing may not provide a complete picture of students' understanding. The Knowledge-Learning-Instruction framework emphasizes that instruction and practice must be tailored to instructional goals, with the possibility of students learning procedural skills well but not being able to explain the skills conceptually [5]. There is thus the possibility that students may write functional code but be unable to trace the execution of their code or the execution of similar code that they did not write. Students' application of their knowledge may also be sensitive to context: as described by the Knowledge in Pieces theory, small variations in the way a question is asked may trigger students to answer differently, using individual elements of a collection of ideas rather than drawing from one unified set of knowledge in the same way across contexts [6]. While this theory of learning has traditionally been applied to the natural sciences, it is more widely applicable to a range of domains [7].

In this paper, we examine intermediate students' understanding of code execution involving multiple Boolean expressions. Specifically, we focus on intermediate students' understanding of code execution when three Boolean expressions are checked in different orders, and if they are checked in a single `if`-statement (conjoined with the logical operator &&) or three nested `if`-statements (each with one condition). Using a survey-based approach, we examine the relation between the code students write (including choice of code structure and ordering of the Boolean expressions), and their code tracing with different structures and ordering of Boolean expressions. This paper makes two contributions:

- A demonstration of intermediate CS students' misunderstandings of code execution. We show that many intermediate students give inaccurate responses to code tracing questions involving incorrect ordering of Boolean expressions, even though the target control structures were taught in introductory classes. While novice pro-

grammers' misconceptions have been studied extensively (e.g., [8]–[11]), intermediate students' ideas have not.
- Evidence that students may show greater proficiency and different kinds of errors when understanding is assessed by having students students write and revise code rather than read others' code. We can interpret this using the theoretical frameworks discussed above. Students' performance suggests that slight changes in code presentation can trigger differences in what knowledge students apply, consistent with Knowledge in Pieces [7]. Large differences in the types of errors made on tasks that ostensibly use the same underlying concepts are evidence for students learning to use certain code structures without understanding why they work, which is suggested by the Knowledge-Learning-Instruction perspective [5].

II. Prior Work

Students' understanding of conjoined conditions and short-circuit evaluation is understudied. Research on misconceptions in programming often focus on novice programmers, and therefore address simpler `if`-statements, with single conditions (e.g., [8]–[11]). Initial work with the Readability and Intelligibility of Code Examples (RICE) survey examined intermediate CS students' comprehension of code that used nested `if`-statements vs. code that conjoined all conditions within one `if`-statement [4]. While those results indicate that intermediate students can comprehend both structures equally, and that they perceive the single `if`-statement as more readable, the particular code examples used conditions that could be checked in any order [4]. A follow-up study with 18 students examined code samples where the ordering of the conditions affected the code execution, finding that the single `if`-statement reduced comprehension [12]. Our study builds on this line of work by replicating the main elements of that study while also incorporating new questions and using a much larger sample size.

III. Methods

This data was collected as part of a study conducted at a large research university in the Mountain West of the United States. The study was conducted at the end of the school semester. Students were recruited from two CS courses, and completed the self-paced assessment questions online. Questions were generally presented individually, and students could not return to previously-answered questions. Of the assessment items relevant for this analysis, all students saw the same questions in the same order. The study design, including data collection plan, and hypotheses were preregistered at OSF (https://osf.io/gzxmf/?pid=74fcw).

A. Participants

Students were recruited from the Data Structures and Algorithms course and an introductory course on Software Engineering (respectively, the second and third programming-intensive courses in the CS major sequence). All of these students received instruction on compound Boolean expressions, including at least brief discussion of short-circuit evaluation, in a prerequisite course on object-oriented programming. Students in Software Engineering were offered extra credit for completing the assessment, and 119 participated (out of approximately 200 enrolled). Since the instructor for the Data Structures and Algorithms class did not want to offer extra credit, students in that class were compensated $5 for participating, and 24 did so (out of approximately 300 enrolled). The assessment had several types of questions (including writing and comprehension). Following our preregistered plan, we excluded from the dataset the 18 students who skipped more than one question of a given type. Of the 125 students included in the dataset, 114 were from Software Engineering and 11 were from Data Structures and Algorithms.

Write a function that takes two ints as input and returns a String. The first and last line are provided for you.

- When the first input divided by the second is 5 or larger, AND the first input is bigger than 10, return the String "Ok".
- Otherwise, return the String "Not Ok".

Hint: Be careful to avoid an exception caused by dividing by 0.

Examples:

Input	Returned
36, 6	"Ok"
6, 36	"Not Ok"
2, 0	"Not Ok"

Fig. 1. Code writing task with sample input-output pairs. The method signature (first line) and closing brace (last line) were provided.

B. Materials

Our assessment questions are adapted from the Readability and Intelligibility of Code Examples (RICE) survey [4], [12]. This analysis focuses on items that target `if`-statements with multiple conditions, specifically examining two factors: the ordering of the conditions (when one order correctly handles edge cases and the other order does not), and if the conditions are conjoined in one `if`-statement or separated into individual `if`-statements. This analysis examines three kinds of questions: (1) writing, (2) revision of the student's own written code in response to feedback, and (3) comprehension. Students saw the writing task first, then the comprehension questions. The survey included automated evaluation of students' written code, and students whose code was flagged for revision saw those revision prompts at the end of the survey.

The writing task (figure 1) explicitly asks students to check two conditions on two inputs (ints): if the quotient is 5 or larger, and if the first input is bigger than 10. A hint reminds students to avoid a divide-by-0 exception in their code. Three sample input-output pairs show the desired functionality, including when the divisor is 0. Students wrote their code in an

open-response text box. There was no functionality to compile or run the code from within the survey. At the end of the survey (after the comprehension questions), students who had not used expert structure were prompted to revise their code.

The survey software allowed for branching based on pattern matching from regular expressions, and this was used to flag students' code on the writing task for later revision. Code was flagged if there was more than one `if`, or if the code was missing the pattern "!=0". If the code was flagged, students were given a set of revision opportunities. The first opportunity presented the student's original code with the original question prompt, with the instructions "Copy your code into the box below and try to improve the code's style. If you see any other errors, please fix them." This instruction did not offer hints or new information, and was intended to measure improvement simply from drawing students' attention back to their code. If the code was still flagged after this revision chance, students were given specific hints. The hints said "We detected at least one of these things about your code: the code has more than one `if`; the code seems like it might throw an exception if the second input is 0 (note, there are a few ways to avoid this exception, and the survey doesn't recognize all of them – just double check)." Students then indicated, with a multiple-choice question, if they could edit their code to follow the hints, if they didn't know how to do so, or if their code was actually fine. Based on that response, there was a final round of branching: students who said they could follow the hints were asked to do so; students who said they could not follow the hints were shown an example for a similar problem (Figure 7); students who indicated that their code was fine were moved on to the next section of the survey.

For the comprehension questions, students were first given a description of the task that the code was intended to accomplish, shown in figure 2. The task involves comparing an inputted character to a character in a specific position in a String. If the given position is out of range, the method should return the input character. Accomplishing this task requires checking the character at the specified position in the String, but must first ensure that the position is valid. The desired behavior for invalid positions is shown with specific examples for positions that are too small and too large (the last two examples in figure 2).

For this task description, students were asked two kinds of comprehension questions: (1) given a code sample, does the code accomplish the task? and (2) given a specific input, what is the output for that code sample? Comprehension questions were presented on different pages that had to be completed in order. Students could not return to previously-answered pages. The first page showed the task description (figure 2), and the first code sample (figure 3), and asked if the code accomplished the task (yes or no). This first code sample checked three conditions (if the position was too small, if the inputted character was greater than the character at the target position in the String, and if the position was too big). However, this code sample checks if the position is too big after attempting to access the character at that position, which

Decide if the code below accomplishes this task:
Task: The function takes as input a String (word) a char (letter) and an int (position), and returns a char.

- Return the letter at the inputted position for the inputted word IF that letter is larger than the inputted letter.
- Otherwise, return the given letter.

Examples:

Input	Desired Return
"Word", 'A', 0	'W'
"Word", 'Z', 0	'Z'
"Word", 'A', -1	'A'
"Word", 'A', 10	'A'

Fig. 2. Task and sample input-output pairs for the code comprehension items.

```
public static char greater5(String word, char letter, int position) {
   if (position > -1 && word.charAt(position) > letter && position < word.length()) {
      return word.charAt(position);
   }
   return letter;
}
```

Fig. 3. The first code sample in this sequence. All conditions are conjoined with && in a single `if`. The target character is accessed before checking if the position is in range, causing an exception when the position is too big.

will cause an exception if the position is indeed too big. Therefore, the code sample does not accomplish the task. The next question page showed the code sample again (figure 3), and asked what the output would be for specific inputs: (1) "Word", 'A', 0 and (2) "Word", 'A', 10. The options for both questions were: 'A', 'W', Something else (with open-response box), and Throws an exception.

The next question page repeated the task description and presented the rest of the code samples (figures 4, 5, and 6), again asking if each sample accomplished the task. The last page of this comprehension set again presented the code samples in figures 4, 5, and 6, and asked what the output would be for the input "Word", 'A', 10. The multiple-choice options were 'A', 'W', Something else (with open-response box), and Throws an exception.

```
public static char greater1(String word, char letter, int position) {
   if (position > -1 && position < word.length() && word.charAt(position) > letter) {
      return word.charAt(position);
   }
   return letter;
}
```

Fig. 4. The second code sample in this sequence. All conditions are conjoined with && in a single `if`. Checking that the position is in range happens before the target character is accessed. Short-circuit evaluation will avoid an exception when the position is out of range.

```
public static char greater2(String word, char letter, int position) {
   if (position < word.length()) {
      if (position > -1) {
         if (word.charAt(position) > letter) {
            return word.charAt(position);
         }
      }
   }
   return letter;
}
```

Fig. 5. The third code sample in this sequence. Each condition is in its own `if`. The ordering of the conditions matches the code in Fig. 4

```
public static char greater3(String word, char letter, int position) {
   if (position > -1) {
      if (word.charAt(position) > letter) {
         if (position < word.length()) {
            return word.charAt(position);
         }
      }
   }
   return letter;
}
```

Fig. 6. The fourth code sample in this sequence. Each condition is in its own `if`. The ordering of the conditions matches the code in Fig. 3, and will throw an exception if the position is too large.

Check out these examples - these three code blocks do the same thing.

```
public static String dontDivideBy0_a(int num1, int num2) {
   if(num2 != 0 && num1/num2 < 25 && num1 + num2 > 15)
      return "Super!";
   return "try again.";
}

public static String dontDivideBy0_b(int num1, int num2) {
   if(num2 == 0)
      return "try again.";
   if(num1/num2 < 25 && num1 + num2 > 15)
      return "Super!";
   return "try again.";
}

public static String dontDivideBy0_c(int num1, int num2) {
   if(num2 != 0) {
      if(num1/num2 < 25)
         if(num1 + num2 > 15)
            return "Super!";
   }
   return "try again.";
}
```

Fig. 7. Code examples shown to students who did not know how to revise their own code to use only one `if` and to avoid a divide-by-0 exception. The example shows three different code structures with the same functionality.

TABLE I
STRUCTURES AND APPROACHES FOR WRITING AND REVISION

	Initial Writing	Revision Chance	Revision 2 w/ Hints	Revision 2 w/ Hints & Examples
No 0-check	57	18	1	0
Has 0-checks:	68	30	71	7
- One `if`	17	15	58	6
- Two `if`s	44	15	12	1
- Three `if`s	2	0	0	0
- No division	1	0	0	0
- Try-catch	4	0	1	0
0-check order:				
- correct	62	28	60	6
- incorrect	1	2	10	1

Note: All 125 students are included for *Initial Writing*. Students who revised their code after it was flagged are included in the subsequent columns. Students had two opportunities to revise (the *Revision Chance* with no new information, and then either a revision after hints alone or after hints and examples). The columns are not cumulative.

TABLE II
USE OF CONJOINING WHEN WRITING CODE

	Initial Writing	Revision Chance	Revision 2 w/ Hints	Revision 2 w/ Hints & Examples
All conjoined	17	15	58	6
Two conjoined	45	15	13	1
No conjoining	1	0	0	0

Note: Entries are counts of code structures. Counts include only those student answers that explicitly checked for zero without using try-catch.

IV. RESULTS

All 125 participants in our sample answered the code-writing question. The comprehension tasks included 9 individual questions (each of the four code samples had a question on overall functionality and output for an edge-case input; the first code sample had one additional question on the output for a non-edge-case input). Three questions were each skipped once (by three different students); otherwise, all students in the sample answered all questions. We discuss response rates to the revision task in that section (below).

A. Writing Results

The writing task required students to check two conditions described explicitly in the problem statement (the size of the first input and the size of the quotient of the first and second input) and one condition referenced in a hint provided with

TABLE III
TREATING THE ZERO-CHECK AS A SPECIAL CASE

	Initial Writing	Revision Chance	Revision 2 w/ Hints	Revision 2 w/ Hints & Examples
Isolated	36	14	10	1
Joined	8	1	2	0

Note: Entries are counts of code structures that included two conditions conjoined but not all three conditions conjoined. Isolated means the check for zero was in its own condition while Joined indicates the check for zero was conjoined with one other condition.

```
if(num2 == 0){
  return "Not Ok";
}
else if(num1/num2 >= 5 && num1 > 35){
  return "Ok";
}
else{
  return "Not Ok";
}
```

Listing 1: Student code writing response, using an *isolated* zero-check. The other two conditions are conjoined with the AND operator. This coding pattern indicates that students can use && to conjoin conditions, but seem to think about the zero-check differently from the other conditions.

the problem (that the divisor was not 0). Of the 125 students, one wrote code that only checked the size of the quotient; 56 checked the two explicit conditions, but did not check if the divisor was 0 (these categories are combined as *No 0-check* in Table I); one checked if the divisor was 0 and the size of the first input (but did not perform the division to check the quotient, *No division* in Table I); and 67 checked all three conditions (or handled exceptions through try-catch blocks). Of the 67 students who checked all three conditions, only 17 of them (14% of our sample) did so with expert structure, conjoining the three conditions within one `if` (*One `if`* in Table I). 46 students checked all three conditions but used more than one `if` (*Two and Three `ifs`* in Table I). Table I also shows that 98% of students who checked that the divisor was 0 and performed the division did so in the correct order (last two lines, including all 63 students who checked for 0, performed the division, and did not use try-catch). This included all students who conjoined all three conditions.

Structure Table II shows three types of structures among the 63 students who addressed all three conditions without try-catch: one expert (all conjoined in one `if`-statement, 17/63), and two novice (one condition per `if`-statement, with *no conjoining* 1/63; and conjoining conditions within `if`-statements, 45/63). The low rate of expert structure is consistent with the preregistered hypotheses. Notably, one of the students who used three `if`-statements did so with conjoined conditions, including an unnecessary check of whether the dividend was 0 in addition to the divisor. Further, three students conjoined conditions while using try-catch, resulting in only 2/67 who checked all conditions but did not use the logical operators AND or OR. Therefore, while novice structures were used by 75% of the 67 students who checked all three conditions, 97% of these students demonstrated the ability to use Boolean operators to conjoin conditions.

Approach. As a whole, the students treated the expression that checked if the divisor was 0 (the *zero-check*) differently from the other two conditions: of the 57 students who only checked two of the three conditions, the zero-check was overwhelmingly the one left out (56/57). Even the students who checked all three conditions approached the zero-check differently than the other two. Among the 47 students who used two `if`-statements and a zero-check, 36 students isolated the zero-check in one `if`-statement and conjoined the other two in another `if`-statement; only 8 conjoined the zero-check with one of the other conditions (*Isolated vs. Joined* in Table III). The prevalence of isolating the zero-check could indicate a desire to separate this "error" case from the more typical cases.

Discussion: Writing Results. While the initial RICE results presented conjoined conditions as an alternative to nested `ifs` [4], these results show that students are more likely to use an in-between structure, conjoining some conditions but not all of them. Therefore, the lack of conjoining all three conditions seems not to be due to a lack of knowledge of the syntax for Boolean operators. While it could stem from unfamiliarity with short circuit execution, that hypothesis is not consistent with the comprehension results (discussed below). Based only on this code generated by the students, we cannot assess whether students believe the zero-check must be separated in this way for functionality or if it is a stylistic choice that they believe enhances the readability of their code. Listing 1 shows one example of a student separating the zero-check. This code, like many of the examples with multiple conditions, uses `if-else` (others use sequential `ifs`). This leads to some code duplication (e.g., `return ''Not Ok''`), but avoids the need to trace through nested `ifs` or to consider the order of execution of conjoined conditions (since the conditions which are conjoined could be evaluated in either order). Again, increased code duplication combined with structures that may be easier for students to trace may be a stylistic choice or a choice driven by uncertainty about how to use alternatives. Overall, these results show that while neglecting the zero-check was a common error (46%, 57/125), and isolating the zero-check was a common strategy (29%, 36/125), the error of mis-ordering the zero-check was extremely rare (1/125).

B. Revision results

Students revised their code at the end of the survey, after the comprehension questions. With chances for revision, a significant number of students were able to revise their code to follow an expert structure, including a check for zero (Table II); students were more successful than expected in the preregistered hypotheses, where we predicted fewer than 40% of students would be successful. When students who wrote non-expert structured code were given the chance to revise (and subsequent hints and examples, if needed), many incorporated a zero-check, and most of them ordered it correctly. Results for the original writing and for the revision phases are show in Table I. For all tables, note that once students wrote expert structured code, they were not prompted to revise further. Also, the tables are not cumulative: the columns for revision show the structures that students used at that time point, and only for students who made some change to their previous code (beyond whitespace). The larger overall numbers for the *Revision 2* columns compared to the *Revision Chance* column show that many students did not revise their

code until they were shown hints for how to do so. 48 revised their code in some way at the first opportunity for revision, when they were not given specific hints. Two of these 48 students submitted code that checked for zero but still could throw an exception due to dividing by zero, either due to placing the check for 0 after the division or using boolean-OR to combine a check for zero and a division (rather than boolean-AND).

At revision 2, when students were explicitly told that one reason their code might have been flagged is that "it might throw an exception if the second input is 0," more students who had initially neglected the zero-check proceeded to incorporate it. While checking for zero at the wrong point in code execution was slightly more common at revision 2 (14% of students, 11/79 submissions), it was still not the norm. One of those 11 students exhibited that error on the initial writing task. Of the other 10, seven of them had not included a zero-check initially. Three students had included a zero-check in their initial writing, and had correctly placed it before the division, albeit in a separate if-statement. These three students introduced the ordering error when they conjoined all of the conditions in one `if`-statement.

Discussion: Revision Results. Many of the initial submissions for the writing question (46%) left out the zero-check, which may have artificially reduced the error of misordering that check. The revision results show that students can incorporate the zero-check, and that most students are not writing code that exhibits the type of ordering error examined in the comprehension questions. Further, given that by Revision 2 many students were revising their code to join all three conditions together (Table II), students appear to know that the check for zero should precede the division: if students were not considering the placement of the check relative to the other conditions, and ordering it randomly, it would result in an ordering error about half of the time. However, it is notable that some students who had the correct ordering of conditions when the zero-check was isolated did not maintain that order when they conjoined it with the other conditions, indicating that they did not realize why that ordering was significant. The writing and revision results alone cannot tell us whether students understand the consequences of alternative placements of the check; students could be sensitive to the placement of the condition when writing their own code while not knowing that this is a functional rather than stylistic necessity.

C. Comprehension results

Comprehension questions presented four methods that each checked three conditions, crossing code structure (conjoined in one `if` vs. separated in individual `if`-statements) with condition ordering (correct ordering to avoid errors by checking that the index is in range before checking the value at that index vs. wrong ordering that does the reverse). As shown in Table IV, student performance on the comprehension questions varied depending on both factors.

Ordering. 91% of students correctly identified that the code would accomplish the task when the code used correct ordering. In contrast, only 45% of students correctly recognized the code would *not* accomplish the task when the code used the wrong ordering (Table IV). This is slightly less than the 54% of students who, for the initial writing submission, wrote code that addressed the edge case of dividing by zero; note, however, that students were explicitly reminded to handle this edge case in the writing question. On the comprehension questions, an edge case was provided with the input-output examples given with the task description, but students were not explicitly told to consider specific errors in the code.

When asked to identify the output for a specific edge case, 89% of students answered correctly for the correct ordering, while only 57% answered correctly for wrong ordering. This suggests that the issue is not simply that students are ignoring the possibility of edge cases. To test the reliability of these differences for each of the two question types, we used logistic regression with mixed effects, with fixed effects for the ordering of the conditions and whether the code separated the conditions in two `if`s or conjoined them, and a random factor for student; the reliability and direction of effects is not impacted by whether an interaction factor is included, and because the models without interaction factors had lower BIC, we do not include the interaction in our analyses. The impact of order was statistically reliable both when students were asked whether the code accomplished the task (coefficient for wrong order: -1.55, $t(497) = 10.6$, $p < .0001$) and when students were asked to identify the output for a given input (coefficient for wrong order: -1.07, $t(497) = 8.15$, $p < .0001$).

Structure. While the impact of whether or not the conditions were conjoined was smaller than the impact of ordering, students were more likely to be correct when the conditions were separated than when conjoined. 74% of students correctly identified whether the code would accomplish the task with separate conditions, compared to 62% when conditions were conjoined (coefficient for conjoined conditions: -0.44, $t(497) = 3.61$, $p < .001$). This difference persisted when students were asked to identify the output for a given input, with 80% of students answering correctly when conditions were separated and 66% answering correctly when conditions were conjoined (coefficient for conjoined conditions: -0.51, $t(497) = 4.31$, $p < .0001$).

Discussion: Comprehension results. Conjoined conditions and incorrect ordering both impaired students' code comprehension. In particular, many students did not recognize that placing a validity check for an index *after* trying to access that index would result in an error. This issue was more pronounced when conditions were conjoined, with only 35% of students correctly identifying that the code did not accomplish the task and 46% of students identifying that the code would raise an error on a given edge case input. Students may not have been considering specific edge cases when asked if the code accomplished the given task. Asking students to identify the output for a specific edge-case input did improve performance

TABLE IV
ACCURACY ON COMPREHENSION QUESTIONS

	Identifying whether code accomplishes the task			Identifying output for a given edge-case input		
	Correct ordering	Wrong ordering	All	Correct ordering	Wrong ordering	All
Conjoined	89% (111/125)	35% (44/125)	62% (155/250)	86% (107/125)	46% (57/125)	66% (164/250)
Separated	93% (116/125)	54% (68/125)	74% (184/250)	93%(116/125)	68% (85/125)	80% (201/250)
All	91% (227/250)	45% (112/250)	68% (339/500)	89% (223/250)	57% (142/250)	73% (365/500)

on the code samples with incorrect ordering. However, the same overall effects of conjoining and ordering persisted. The pattern of errors is not consistent with a straightforward lack of knowledge of short-circuit evaluation. If students simply did not recognize short-circuit evaluation, they would have indicated that both code samples with conjoined conditions would result in errors.

V. COMBINED DISCUSSION

The pattern of results across the writing task and the comprehension task suggests that students' understanding of code execution with Boolean expressions is very context-sensitive. In particular, students were very unlikely to mis-order conditions in the initial writing task (1 did so, out of 63 who included a zero-check). Ironically, incidence of this error rose when students were given hints to improve their code. However, even the 14% rate of students who wrote with that error on Revision 2 is much less than the 43% of students who did not understand the impact of that error on the input-output comprehension task. Students seem to avoid this error when they write, but without a full understanding of what exactly they are avoiding.

When writing code, students may order it in a particular way based on their ideas of functionality, readability, or both. If students believe that the ordering is mainly about readability rather than functionality, they may order their code correctly without considering the impact of order on functionality at all. In that case, students would write correctly, but would not necessary comprehend code that was ordered differently. This indicates that writing code is not a complete assessment, and additional tasks like code reading are needed to fully assess students' code comprehension. Further, if students don't make this error themselves, they may be missing practice opportunities for recognizing it and fixing it.

The high degree of context-sensitivity and the seeming inconsistency between students' actions in different contexts are aligned with the Knowledge-in-Pieces (KiP) framework [7]. In particular, KiP posits that different elements of students' mental models are triggered by different contexts (e.g., a physics student may say that a ball thrown in the air only has the force of gravity acting on it, but when asked about the peak of the toss, invents a second force) [7]. In applying KiP to a programming context, Lewis emphasizes the importance of students' growing realization of what they should be paying attention to as a factor that mediates learning (e.g., attention to state while debugging) [13]. Within this framework, while students are writing, they may be focusing on their own categorization of different types of conditions. Specifically, students may characterize conditions as "regular" checks or "error" checks while they write. However, if this categorization is not conceptually connected to an understanding of code execution, it will not be useful during code comprehension.

A catch-all for flawed mental models of program execution is the "superbug" that the computer will make intelligent choices, manifesting here as the implicit belief that the conditions will be evaluated in an appropriate order [14]. Alternatively, students might have a flawed mental model that doesn't rely on smart ordering of execution but focuses on the underlying logic of a condition rather than how it will be executed. When all conditions are joined with Boolean-AND, one condition being false will make the whole statement false. Reading the code in search of a false condition, ignoring the possibility of an error interrupting program execution, is consistent with students' pattern of accuracy on the comprehension questions. In this hypothesis, student choices are consistent with an incorrect mental model of the notional machine that executes their code [2]. In particular, this error pertains to *when code is evaluated*, a challenging issue for students across languages. For example, when tracing code in Lisp, students must consider whether arguments are evaluated when they are passed or only when actually used; in most contexts, each argument is evaluated before being passed to a function, but for special forms, a different evaluation pattern occurs. Because Lisp is a functional programming language, misunderstandings about evaluation cannot be easily traced to line ordering, and some work has found that explicit practice that highlights the evaluation model can improve understanding and lead to greater proficiency in programming [15]. Our results suggest that the challenges of understanding the evaluation model are not limited to functional programming languages, and explicit instruction on when each part of the code is evaluated might be beneficial even in imperative and object-oriented programming paradigms.

The high accuracy of student responses for correctly-ordered conditions demonstrates that students were able to identify the overall purpose of the code and perform some code tracing. However, the low accuracy on the comprehension questions that raised errors and the differences in accuracy between separated and conjoined conditions both suggest that students may be missing important details about code execution even by the time they reach intermediate CS classes. While many instructors include code reading in their classes, they are likely to assume that students have mastered `if`-statements and Boolean conjoining by the end of the first CS course,

meaning it may come up in examples but is unlikely to be the core targeted concept in activities in later courses. Yet, our results suggest a need for more emphasis on these details of code execution.

One of the challenges of both conjoined conditions and nested `if` statements is that the order of execution is not strictly linear. While there has been limited study of students' comprehension of nested `if`s, Wiegand et al. investigated introductory students' performance on identifying the output of nested `if`s and tracing their execution [16]. Surprisingly, students performed better when identifying the output of a nested `if` statement compared to an `if-else` statement, which Wiegand et al. attributed to challenges with the control flow not being strictly linear: students had to ignore the `else` when predicting the output for the `if` (given the particular questions asked). However, Wiegand et al. also found that tracing nested `if` statements was challenging and suggested that the correct identification of output for nested `if`s might be due to the answers being relatively easy to guess. Our work provides further evidence that tracing and identifying output for at least some types of nested `if` statements is challenging, and the differences in accuracy between questions with the same code structure is evidence that understanding for nested `if`s may not be monolithic and may be overestimated based on features of the assessment.

VI. Implications for Instructional Design

Interpreting these results using the lens of the Knowledge-Learning-Instruction (KLI) framework [5] suggests instructional implications. KLI breaks student knowledge into small chunks, known as *knowledge components*. Writing code versus comprehending code may rely on different knowledge components, explaining why students avoided ordering errors in their writing but were tripped up by them in code comprehension. The KLI framework posits that knowledge components consist of an action and a condition of applicability [5], such as *when code requires an error check, put it first*. This kind of decision can be tacit, meaning that students may perform actions consistent with a hypothesized knowledge component without being able to explain why they are doing it. Instruction, therefore, should focus on the knowledge components needed for code comprehension in addition to those for code writing.

Instructional strategies could include more practice with code tracing, especially with examples where the conditions are ordered incorrectly. This could be appropriate in an introductory class when students first learn about compound logical expressions, but also in a data structures class when students need to perform a null check before some other operation. Emphasizing the topic again in authentic data structures contexts may help students recognize the applicability of the topic for their own code. Instruction can specifically highlight the importance of the correct ordering, and the distinctions among how the code executes when the conditions are in different orders. Further, these results suggest that making sense of the specific order of execution for both correct and incorrect code may be an important instructional activity, especially because students may write correctly without understanding what makes their code correct.

VII. Limitations

Several limitations are inherent in the study methodology. The survey was explicitly a research instrument, and compensation or extra credit were not based on performance. It is possible that the survey results, overall, may underestimate students' abilities if they were not motivated to try hard. Further, the survey platform did not allow students to compile or test their written code. Results from the code writing and revision task, therefore, will not be representative of students' performance when they have such tools. While we interpret the frequency of the isolated zero-check to indicate that students think about it differently than the other conditions, we cannot confirm this interpretation without direct explanations from the students. Similarly, while the pattern of comprehension difficulties indicates that both condition order and structure affect comprehension, our data cannot confirm students' thinking about code execution in each case. Finally, this study does not firmly establish condition ordering as a difficulty that affects code writing or code comprehension outside of a research setting. Future work should: elicit student explanations for their code writing choices; examine students' mental models of code execution; and determine what kinds of misunderstandings around code execution affect students' learning or their performance on real-world tasks.

VIII. Conclusion

This paper presents a study of 125 intermediate computer science students and their writing and comprehension of code that checks multiple Boolean conditions, in cases where the ordering of those conditions can cause or avoid errors. Comprehension tasks examine two factors, and reveal that both the ordering of the conditions and the structure of the code (conjoined conditions vs. nested `if`s) affect comprehension. Specifically, it is much harder for students to identify what the code does, both in general and for a given input, when the ordering of conditions causes errors on edge cases. Notably, this is an error that rarely came up in the code-writing task, indicating that students may use correct coding practices in their writing without fully understanding why they are necessary. In particular, this identified comprehension difficulty may be representative of a larger category: the order of execution within a single line of code. This work demonstrates that even intermediate students in their second and third programming-intensive classes may not fully understand these nuances of execution, and that proficiency with code-writing is not an accurate predictor of code comprehension. Further, this work suggests that students would benefit from instruction around errors, even if they are unlikely to make them, because identifying and explaining them can illuminate students' own mental models of code execution.

References

[1] B. Du Boulay, "Some difficulties of learning to program," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986.

[2] J. Sorva, "Notional machines and introductory programming education." *ACM Transactions on Computing Education*, vol. 13, no. 2, p. n2, 2013.

[3] J. Whalley, T. Clear, P. Robbins, and E. Thompson, "Salient elements in novice solutions to code writing problems," *Conferences in Research and Practice in Information Technology Series*, vol. 114, pp. 37–45, 2011.

[4] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, 2019.

[5] K. R. Koedinger, A. T. Corbett, and C. Perfetti, "The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning," *Cognitive science*, vol. 36, no. 5, pp. 757–798, 2012.

[6] A. A. diSessa, "Toward an epistemology of physics," *Cognition and Instruction*, vol. 10, no. 2/3, pp. 105–225, 1993. [Online]. Available: http://www.jstor.org/stable/3233725

[7] A A. diSessa, "A friendly introduction to "knowledge in pieces": Modeling types of knowledge and their roles in learning," in *Invited Lectures from the 13th International Congress on Mathematical Education*, G. Kaiser, H. Forgasz, M. Graven, A. Kuzniak, E. Simmt, and B. Xu, Eds. Springer, Cham, 2018, pp. 65–84.

[8] P. Bayman and R. E. Mayer, "A diagnosis of beginning programmers' misconceptions of basic programming statements," *Commun. ACM*, vol. 26, no. 9, p. 677–679, Sep. 1983. [Online]. Available: https://doi.org/10.1145/358172.358408

[9] Y. Qian, S. Hambrusch, A. Yadav, S. Gretter, and Y. Li, "Teachers' perceptions of student misconceptions in introductory programming," *Journal of Educational Computing Research*, vol. 58, no. 2, pp. 364–397, 2020. [Online]. Available: https://doi.org/10.1177/0735633119845413

[10] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '03. New York, NY, USA: ACM, 2003, pp. 153–156. [Online]. Available: http://doi.acm.org/10.1145/611892.611956

[11] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3077618

[12] E. S. Wiese, A. N. Rafferty, D. M. Kopta, and J. M. Anderson, "Replicating novices' struggles with coding style," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 13–18.

[13] C. M. Lewis, "The importance of students' attention to program state: A case study of debugging behavior," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 127–134. [Online]. Available: https://doi.org/10.1145/2361276.2361301

[14] R. D. Pea, "Language-independent conceptual "bugs" in novice programming," *Journal of educational computing research*, vol. 2, no. 1, pp. 25–36, 1986.

[15] L. M. Mann, M. C. Linn, and M. Clancy, "Can tracing tools contribute to programming proficiency? the lisp evaluation modeler," *Interactive Learning Environments*, vol. 4, no. 1, pp. 096–113, 1994.

[16] R. P. Wiegand, A. Bucci, A. N. Kumar, J. L. Albert, and A. Gaspar, "A data-driven analysis of informatively hard concepts in introductory programming," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 370–375.