Noname manuscript No.
(will be inserted by the editor)

Tracing vs. Comprehension: On Different Levels of Understanding Boolean Expressions

Aviad Baron · Dror G. Feitelson

Received: date / Accepted: date

Abstract Reading and understanding existing code is a crucial part of software engineering. We focus on the understanding of Boolean expressions, which control the flow of the execution. Such expressions can often be written in different ways that are logically equivalent, but there has been no systematic research on the possible advantages of one formulation over another. Our goal is to examine the effect of various factors on understanding such expressions, leading to guidelines for selecting the most understandable version from a set of logically equivalent expressions.

To achieve this goal, we conducted a controlled experiment using all 16 simple Boolean expressions with two variables, a connecting operator, and possible negations. We define and empirically compare two distinct levels of understanding these expressions: *tracing*, which involves following the program's execution for a specific input, and *comprehension*, which encompasses a generalization of how the code behaves for all possible inputs. The experiment involved 362 participants, 57% of which had more than 6 years of programming experience.

The results reveal that comprehension not only takes longer but also leads to a higher error rate compared to tracing. Furthermore, expressions involving the logical operator OR were found to be more challenging on average than those with AND, but this difficulty manifested only at the comprehension level. One of the sources of this difference appears to be an interaction between the

Aviad Baron
Department of Computer Science
The Hebrew University of Jerusalem, Jerusalem 91904, Israel
E-mail: aviad.baron@mail.huji.ac.il

Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem, Jerusalem 91904, Israel
E-mail: feit@cs.huji.ac.il

logical operators OR and NOT, and we discuss possible models which may explain this effect.

The observed differences in understanding equivalent expressions highlight the importance of selecting which expression to use. Our findings suggest that Boolean expressions with AND and fewer negations tend to improve code readability, making them preferable for writing understandable code. But these recommendations need to be verified in the general context of Boolean expressions, including those that are more complex than the ones we considered.

1 Introduction

Understanding code is a crucial skill for software development, serving as a foundation for writing, debugging, and maintaining code bases [40, 58]. Boolean expressions, used to control branch points in code, are one of the factors which contribute to the complexity of code. For example, McCabe's Cyclomatic Complexity metric essentially just counts such branch points [39]. Given their prevalence and impact, a good understanding of Boolean expressions is essential for efficient code comprehension and manipulation.

One of the elements of Boolean expressions that may make them more difficult to understand is negations. Negation also exists in natural language, and numerous studies have examined how sentences with negations and logical operators are processed in the brain, often employing advanced tools such as fMRI [24, 18, 25, 32, 42, 53, 54, 34, 1]. But the comprehension of Boolean conditions in code remains underexplored in the software engineering literature.

A basic issue in program comprehension research is how to establish that developers indeed understand the code. Conceptually, understanding implies the capacity to respond in a meaningful manner to an instruction or question. Feitelson [22] identifies multiple levels of understanding relevant to code, and distinct tasks that bear witness to the achieved understanding. Two important levels are **tracing** (or interpretation) and **comprehension**. Tracing refers to the ability to follow the code's execution and *predict its outcome*, such as its printed results for a certain input. This reflects a grasp of the individual instructions comprising the code and their cumulative effect. Comprehension, in contradistinction, requires a higher level of abstraction: that of *grasping the functionality* of the code. As such it is not related to any specific input. But how does one compare these levels empirically? Most of the studies listed below as related work operationalize understanding only at a basic tracing level. We know of no studies that examine Boolean conditions across different levels of understanding, and only few that provide any insights comparing these levels.

In our study we fill this gap by considering the understanding of basic Boolean expressions at both levels. We operationalize the tracing level as determining whether a condition holds for a given specific input. We operationalize the higher level of comprehension as the ability to generalize the Boolean condition's behavior across all possible inputs, effectively understanding for which input classes the condition holds and for which it does not. This is ap-

propriate when we limit the discussion to basic Boolean expressions that do not have any independent functionality. To test the understanding of logical conditions at these two levels we conducted an experiment comprising two distinct parts, with each part evaluating one of the levels.

Quite surprisingly, we know of no previous experiments which have investigated this issue. The use of De Morgan's laws is well-known in the simplification of logic circuits, where they are used for the expression of any given circuit using only NAND gates [56]. But it is not known which of a pair of equivalent logic formulas is easier for *humans* to understand, or even whether such a difference exists at all. Our work appears to be the first to systematically explore this issue. As such, it is limited to certain simple cases, where many extraneous factors have been eliminated. Much additional work will be needed to investigate how our results apply to the more general case of arbitrary Boolean expressions that may be used in real programs.

Our contributions in this paper are:

- We suggest a concrete definition of the distinction between tracing and comprehension as being able to find the result of the code for a specific input (tracing) vs. characterizing the results for all inputs (comprehension).
- We establish that comprehending simple Boolean expressions is harder than tracing, in that it takes longer and suffers from more errors. It is also not amenable to a simple model based on the expression's syntax, while the time to perform tracing does have such a model.
- We discover that understanding simple expressions with the OR operator is generally harder than understanding such expressions with the AND operator, but only at the comprehension level. No such difference is observed on average at the tracing level.
- We identify cognitive factors that cause difficulties in expressions with OR, particularly the interaction between the logical operators NOT and OR.
- We compare different ways of writing logically equivalent Boolean conditions in code, and recommend conditions using AND and with fewer negations be used to improve readability.
- We investigate the tendency to short-circuit when reading expressions during code tracing, revealing some use of short-circuits in conditions with the OR logical operator compared to none in conditions with the AND operator.
- We note that the differences between comprehension levels imply that experiments on code comprehension need to distinguish the levels and select the appropriate ones. The results from one comprehension level cannot be generalized to another.

All experimental materials are available on Zenodo for further perusal[1].

[1] https://doi.org/10.5281/zenodo.14970257

2 Related Work

Programming is hard and intellectually challenging [20,44]. Numerous studies have looked into what makes it hard, especially from the perspective of teaching programming to novices [8,48,51,49]. Many have focused on specific programming constructs, such as loops and recursion [31,50,33,6,27,3]. Suggestions for code complexity metrics have been based on counting such constructs [39,10]. In particular, this has included the logical operators used to create compound logical expressions [39,16].

However, few studies have directly addressed the understanding of logical conditions. Ebrahimi found that problems pertaining to the understanding of conditionals may be attributed to a failure to appreciate the difference between AND and OR in if statements [21]. His interpretation was that these operators are mistakenly understood as they are in the English language. A similar sentiment was also expressed by others, mainly in relation to the OR operator. According to Herman et al., students tend to misinterpret the OR operator as true when one of the operands is true, but not both, because that is the common way to use "or" in English [28]. However, they also noted that OR and XOR, together with AND and NOR, are simple operators that students intuitively understand correctly. Grover and Basu investigated the comprehension of Boolean conditions among students using the Scratch programming environment [26]. They also found confusion regarding the logical operator OR, which many participants interpreted as XOR, and gave the same explanation that this is how "or" is used in English.

Focusing on Boolean expressions involving negations, Iselin performed an experiment on understanding loops with a condition that either did or did not include a negation, finding that positive conditions are easier to process than negative ones [31]. The study by Herman et al. cited above also included an investigation of students' difficulties in writing Boolean expressions [28]. One was a tendency to omit negated variables, e.g. when a recipe says "use cinnamon by itself" they added the variable representing cinnamon, but forgot the negated variables representing other possible ingredients. Chen et al. developed a testing method for Boolean expressions that finds, inter alia, wrong negations [13]. Ajami et al. found a difference between equivalent conditions with and without negations, but did not reach a conclusion about the precise factors that cause the difference [2]. Baron et al. explored Boolean expressions [4], uncovering that processing negations can pose challenges, and that regularity in expressions significantly eases comprehension. Several online resources also suggest that developers should avoid negations, and especially double negations, e.g. [12,41,35]. But these are not based on systematic studies (and may contain basic mistakes, such as one that presents the code if (!isCar || !isElectric || !isFast || !isAwesome) {isTesla = false}, and then claims that it is equivalent to the English "It is not a Tesla if it is not an electric car, and it's not fast and it's not awesome", substituting AND for OR). We know of no study that systematically compares AND and OR with negations as we do.

Developers spend much more time reading existing source code than writing new code [40,58]. Program comprehension is therefore an important aspect of software engineering. As noted above, Boolean expression are just one type of construct that can make code harder to understand. Other factors may include code length, code layout, syntactic structures, variable naming conventions, and other structural and semantic elements that shape how code is perceived in the past 40 years and processed by programmers [23,19,9,11]. A significant body of research has been performed in the past 40 years on the factors influencing the difficulty of understanding code [7,57].

Concerning the levels of understanding programs, many studies have used tracing (and specifically, determining what a program will print) as an indication of understanding, e.g. [2,5,43]. As noted above, other definitions of understanding were suggested by Feitelson [22], but most were rarely if ever used in actual studies. Hurtig et al. claim that learning to think symbolically about conditionals is hard for students, and use this to test a tool by which educators can follow the learning process of students [30]. This directly relates to our distinction between finding the outcome for a specific input and inferring the behavior for all inputs. We know of no other study that directly compares the understanding of specific constructs at different levels, nor any which use Venn diagrams to assess comprehension as we do (as described below).

However, a series of papers by Lopez, Lister, Teague, and co-workers suggests that the different levels of understanding may be pre-requisites for learning to program. In their first paper, they considered multiple levels of knowledge of code, including data types, tracing simple and iterative code, and understanding code (which they represent by questions of "explain in plain English" what the code does, emphasizing that this does not mean "giving a description of what each line does") [38]. In addition, they tested the students' ability to write code to a specification. The main results indicated that explaining code was related to being able to trace iterative code, and that writing code was related to both tracing iterative code and explaining code. Based on these results they suggested a theory that learning to write code requires a hierarchy of skills to be mastered. A subsequent paper reproduced these results using a different cohort of students from a different university, learning to program in the different programming language, and using a different analysis methodology, thereby significantly boosting the perceived validity of the original study conclusions [37].

A third paper emphasized the difference between tracing and explaining, showing that some students who can trace code can not explain it at a more general level [52]. This distinction was shown to be consistent with a neo-Piagetian stage theory of programming. Specifically, the ability to trace code reflects the preoperational stage, where students are not yet able to reason about the code's purpose. This comes only at the concrete operational stage, which corresponds to being able to explain the code. Several additional papers delved deeper into this theory and its implications for teaching programming [36,15].

The difference between this line of work and our work is that they are concerned with how complete novices learn to program, and suggest that learning to write code requires one to first acquire the skills of understanding code. We are concerned with the understanding of simple Boolean expressions by seasoned programmers, not novices. It is therefore interesting to observe that in both these contexts there appears to be a distinction between tracing code and fully understanding its purpose.

3 Research Questions

Our experiment focuses on comparing the understanding of short code snippets that involve various logical expressions, including expressions with and without logical negation, and equivalent expressions. In this context, our research questions are as follows.

RQ1 **Is there a difference between tracing and comprehension as indicators of understanding code?** While it has been speculated that such a difference may exist, it appears that this question has never been investigated empirically to ascertain whether this is indeed so. As this is a very broad question, we start with the concrete case of understanding basic Boolean expressions. We define understanding at the tracing level as the ability to determine the truth value of a Boolean expression for a specific input instance, and comprehension as the generalization across all possible inputs.

RQ2 **Is there a difference in the difficulty to understand Boolean expressions using AND and OR?** Prior research has suggested that such differences are small and depend on interactions with other factors [4]. But these results were limited to using tracing, so they may just reflect the fact that syntactically AND and OR are indeed very similar. They may differ, however, when a higher level of comprehension is considered—and our results indicate that in fact they do.

RQ3 **Are there differences in the difficulty to understand different Boolean expressions that are actually semantically equivalent?** In other words, does the structure impact understanding, or is it only the semantics that matter? And does this depend on the level of understanding? This question has important practical implications, as it may lead to guidelines concerning the choice of more understandable formulations.

RQ4 **Do developers use their understanding of Boolean expressions to "short-circuit"?** Short-circuiting refer to the possibility of determining the outcome without considering the full expression. For example, in an expression with two literals[2] connected by the logical AND operator, if the first literal evaluates to FALSE, there is no need to check the truth

[2] Following the terminology used in propositional logic, a "variable" is defined to be a Boolean variable, namely an atom that can be TRUE or FALSE. A "literal" is defined to be a variable or a negated variable.

Table 1 The 16 basic expressions used in the experiment.

$p \wedge q$	$p \vee q$	$\neg(p \wedge q)$	$\neg(p \vee q)$
$\neg p \wedge q$	$\neg p \vee q$	$\neg(\neg p \wedge q)$	$\neg(\neg p \vee q)$
$p \wedge \neg q$	$p \vee \neg q$	$\neg(p \wedge \neg q)$	$\neg(p \vee \neg q)$
$\neg p \wedge \neg q$	$\neg p \vee \neg q$	$\neg(\neg p \wedge \neg q)$	$\neg(\neg p \vee \neg q)$

value of the second literal. Employing such reading shortcuts can make tracing faster, but depends on the structure of the expression.

In our present exploration of all these questions, understanding is defined as finding the output a code snippet prints (in the context of tracing), or for what inputs it prints a certain output (in the context of comprehension). Difficulty is measured by the time this took and the fraction of wrong answers [45]. The questions of whether the results depend on these choices are left for future work.

4 Experimental Design and Execution

The experiment was designed to explore the understanding of Boolean expressions, and specifically the effect of their most basic building blocks: the operators AND, OR, and NOT. It was based on 16 basic logical expressions with 2 variables, shown in Table 1. This is the complete set of expressions that can be built with 2 variables, a connecting operator, and negations. Note that this leads to 8 pairs of equivalent expressions related by De Morgan's laws.

There are two reasons for focusing on these simple expressions. The first is that most Boolean expressions in real code are rather simple, often containing only a comparison (e.g. `if (len == n)`). Thus simple expressions are actually more representative than complex expressions with multiple operators and nesting[3]. Second, as the issue of comparing levels of understanding of Boolean expression has not been studied empirically before, it is necessary to focus on just this factor and exclude all others. This is best done using simple expressions, with as little surrounding code as possible. It allows us to focus on the effect of the logic itself, and exclude potential confounding interactions with other code elements.

The experiment was divided into two main parts. The first part comprised 64 questions, 4 for each code snippet, with each participant receiving 16 out of the 64. This section focused on trace-based understanding, where participants were presented with Boolean conditions and a visual input, and were asked to determine what the code would print for the given input. The second part of the experiment dealt with deeper comprehension of Boolean conditions. It included 16 questions in total, with each participant receiving 8 of them. Each

[3] In a separate unpublished research project, based on analyzing 350 thousand Boolean expressions from 1000 GitHub projects written in Python, we found that 80% of the expressions used in branching constructs had only one operator, which could be either a logical operator or a comparison operator.

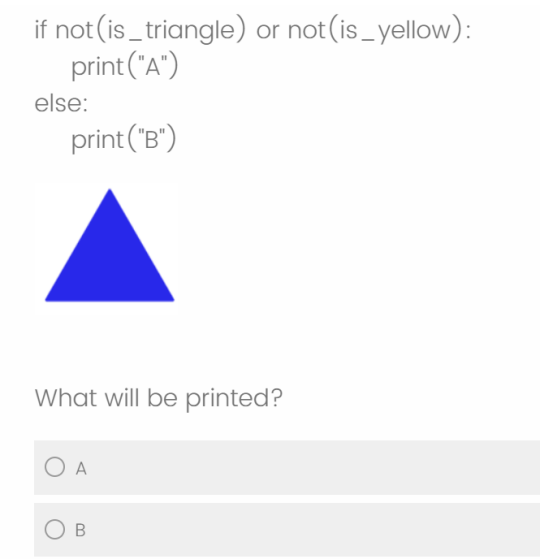

Fig. 1 Example question from part 1 of the experiment.

question featured a code snippet with a Boolean condition, accompanied by a Venn diagram, and participants were asked to determine how the expression would evaluate for all possible inputs as reflected in the diagram.

The variables in the expressions related to geometrical shapes and colors (e.g. is_square instead of p in Table 1). This was chosen as a neutral domain that does not invoke any preconceptions and is readily understood by anyone.

4.1 Part 1: Tracing

The first part centered on understanding Boolean expressions through tracing. Each expression shown to participants was followed by a visual input, and the participants were asked to determine the output of the expression for this input. Each expression appeared in four questions, with four different inputs: one where both variables are assigned the value TRUE, one where both variables are assigned FALSE, and two depicting cases where one variables is TRUE and the other FALSE. An example is shown in Figure 1. In this example, the correct answer is "A", because the shape is not yellow.

The reasons for this design were as follows. Using a visual input was done to highlight the interaction with the truth values of the variables. If we had initialized each variable's value directly as TRUE or FALSE, there would have been no processing of the truth value. Consequently, interactions such as those between logical negation and the truth value of the variable would have been less pronounced. On the other hand, we did not want to create variables whose truth values would be complex to understand, as this could introduce significant bias. Such complexity would shift the experiment's focus toward the

difficulty of processing the variable's truth value in a specific context, rather than examining the logical condition itself in the clearest possible manner—specifically, its structural form and the interaction between its logical operators. Therefore, using a visual representation that made it easy to discern the truth value of each variable, while still requiring some level of processing, provided a good balance.

A second reason was our desire to offer a specific input experience, akin to real-world scenarios, rather than hardcoding the initialization of the variables. Initializing variables would have resulted in more artificial code, since in real coding scenarios the code is designed to handle various inputs, and we do not know the specific input values ahead of time. By providing a visual input, we emulate the assignment of inputs to variables, and illustrate that the code is not tied to a specific input instance. This approach reduced artificiality and made the code more general.

Finally, we also wanted to avoid the possible issue of initialization order. If we had initialized the truth values of the variables, the order of initialization could perhaps also be a confounding factor that influences the process of understanding.

A possible problem with using a visual input is that this appears to require additional cognitive processing, which is unrelated to reading and understanding the code. This could introduce a bias to the results. But according to dual coding theory this is not necessarily the case.

Dual coding theory is based on the premise that comprehension is achieved by two cognitive systems operating in parallel: the verbal system and the non-verbal system [47,46,14]. The verbal system is concerned with parsing and understanding language, whether written or spoken, including reading program code. It is sequential by nature. The non-verbal system aids comprehension by using images and other modalities, such as sounds or smells, either sensed or generated internally (e.g. mental images or memories).

Providing the input in a visual form, as we did, activates the non-verbal system, which interacts with the processing of the Boolean expression by the verbal system, and provides a concrete setting for it. According to the theory, this is expected to enhance the cognitive processing. If the input were to be given in the form of initializing Boolean variables with a visual interpretation (e.g. is_triangle = True), the non-verbal system would be activated anyway to create the same mental image. Thus we do not expect the added modality of having a visual input to cause any problems, especially given that recognizing basic shapes and colors is extremely trivial.

4.2 Part 2: Comprehension

The second part of the experiment aimed to assess a deeper understanding of logical conditions in code, specifically the comprehension of their meaning. We define this deeper understanding as the developer's ability to grasp what the expression would return for *all* possible inputs. In other words, it refers to the

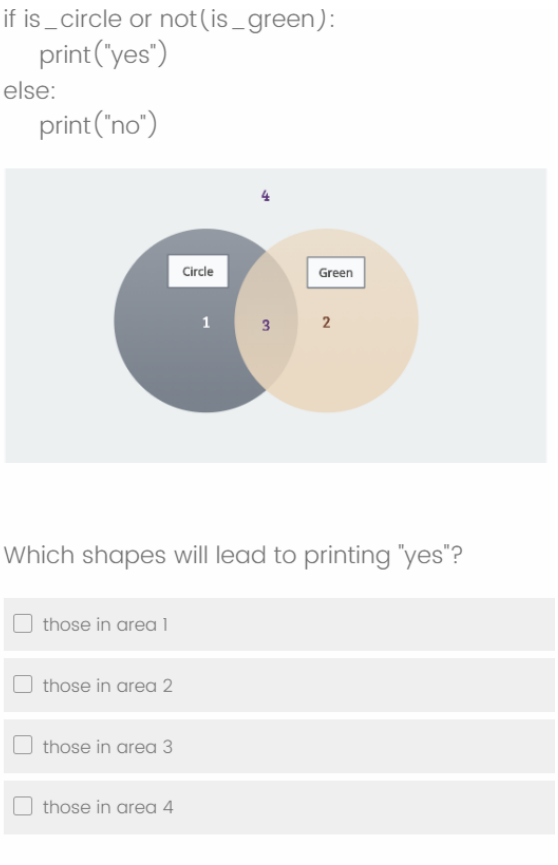

Fig. 2 Example question from part 2 of the experiment.

capacity to determine for which inputs the expression returns TRUE and for which it returns FALSE. This level of understanding reflects a higher cognitive process, as the developer is not merely following the processing of a specific input, but has grasped the deeper logic behind the expression. It demonstrates an understanding of the semantic meaning of the expression, rather than just its syntactic structure.

In this part of the experiment, participants were presented with a Boolean condition alongside a Venn diagram question related to the condition. They were asked to determine for which regions in the diagram the condition holds true. The visual representation of the areas in the diagram was consistent for all the expressions, and used distinct colors from the colors named by any of the variables. This was to avoid confusing or leading the participants.

The code snippets in this part featured the same 16 expressions that appeared in Part 1. In Part 1 we had 64 questions, because we generated an input for every possible assignment of the variables. In this part, there was no spe-

cific input, as we were assessing a more generalized understanding. Therefore, there was only a single question for each expression. An example question is shown in Figure 2. In this example the correct answer is to mark areas 1, 3, and 4 (note that in this part of the experiments the questions are multiple choice, as was also clarified in the experiment instructions). The first clause, is_circle, is TRUE for areas 1 and 3. The second, not(is_green), is TRUE for areas 1 and 4. As they are connected by an OR, the final answer is the union of these two sets. The Venn diagram was always rendered in the same generic way — it was intentionally not adjusted to reflect the semantics of the expressions (e.g. coloring an area green if it represents green shapes) to retain its status as an abstract representation of the input space.

The considerations for creating a visual representation using a Venn diagram were as follows. First, had we expressed the answers in long textual form describing different objects, the structure of such expressions could have influenced the results due to similarities between some possible answers and the way the logical condition itself is phrased. This resemblance could have introduced a bias, leading participants to provide answers not based on real comprehension of the expression's meaning, but rather on syntactic similarities. Therefore, using the visual tool of a Venn diagram helped distance the task from such linguistic similarities, enabling us to assess the participants' internalization of the logical expression's meaning more effectively, without this type of bias.

Moreover, crafting verbal answers of varying lengths, instead of concise uniform-length responses, could have introduced additional bias, which we also sought to avoid. Using the Venn diagram facilitated the use of multiple-choice questions, that can also be graded automatically.

We acknowledge that employing diagrams in this way is uncommon, and may therefore create some "cognitive overhead" for the participants, biasing the results. There is a danger that participants might need to invest mental effort in understanding the diagram, in addition to the mental effort they invest in understanding the Boolean expression. However, we believe that in the balance this is a good way to test a more abstract level of understanding. We expand on this issue in the discussion of the results.

4.3 Experiment Design

In both parts of the experiment, participants received only a subset of the code snippet questions. This was done to avoid less natural and overly focused reading of the expressions due to familiarity and habitual responses. It also served practical purposes by simplifying the experiment and preventing it from becoming too lengthy. Each participant was randomly assigned a subset of the code snippets, and the order in which they were presented was also randomized to eliminate any bias related to the sequence of appearance. In the first part, participants were randomly assigned 16 of the 64 questions. In the second part, participants were randomly assigned 8 out of the 16 questions.

Note that due to the randomization in assigning questions, each participant may not receive the same expressions in both parts of the experiment. They may also not receive pairs of equivalent expressions. As a result comparisons of results for different expressions are between subjects, and we can not factor out individual differences between participants.

The parts of the study were designed to be separate from one another, as our primary goal was to examine the cognitive processes and internalization of the various expressions. We wanted to avoid any potential mixing that could influence different types of understanding. When each section consists solely of similar question types, the thought processes regarding the expressions become consistent and uniform, providing a clearer reflection of the analysis and comprehension for each type.

In addition, we did not want to position the comprehension section first, before the tracing part, because the deeper understanding it entails incorporates the understanding developed in the tracing phase. In other words, we wanted the first part to represent the more fundamental level of understanding. We suspect that anyone who understood an expression as required in the comprehension part would also grasp it adequately for the tracing part. Therefore, we wanted to avoid the danger that participants may develop semantic understanding and use it during the tracing, as may happen if the comprehension questions came first. In contrast, transitioning from the tracing part to the comprehension part would not allow for the continuation of the same understanding approach, as responding to the questions in the second part necessitates a comprehension that surpasses what is required in the first part.

We recognize that by the time participants reach the second part, they may be generally more tuned to reading logical expressions. However, if the experiment reveals that the second part is significantly more difficult than the first, this would only strengthen our findings.

In both parts of the study, there was an introductory slide before the actual questions. Before the first part this slide explained that the section would include code with logical expressions, a visual input, and a question regarding what the code would print for that input. In the second part, there was a detailed explanation of the Venn diagram, including the numbering of the areas within it, and what each numbered area represented. The numbering was done using numerals rather than letters to prevent confusion with the standard notation in set theory, where groups are typically denoted by letters such as A, B, etc.

The experiment ended with 4 demographic questions about gender, education, experience, and current occupation. These allow for a characterization of the participants. The whole study received ethics approval from The School of Computer Science & Engineering Committee for the Use of Human Subjects in Research.

Table 2 Demographics of experiment participants.

Total participants	362
Answered gender question	141
male	126
female	6
other	9
Answered programming background question	141
from high school	11
self taught, vocational training, or army	19
current BSc student	10
BSc degree	58
MSC student or degree	35
PhD student or degree	8
Answered occupation question	141
academic career / student	23
professional developer	106
other	12
Answered question on real programming experience	135
[0-2] years	17
(2-6] years	41
>6 years	77

4.4 Experiment Execution

The experiment began with an introductory page outlining its purpose and structure. This included details about the number of questions, the estimated duration, and an overview of the experiment's objective: "Our goal is to understand the cognitive mechanisms involved in reading various logical expressions in code." Participants were also informed that the experiment would measure their response times, and they were instructed to respond only when fully focused and free from distractions. Consent to participate was obtained by a statement indicating that advancing to the questions themselves implies such consent. Participation was completely anonymous, and no identifying information was collected. Participants did not receive any compensation for their participation.

The experiment was conducted through the Qualtrics survey platform. Invitations were disseminated via WhatsApp groups for developers, as well as developer forums on Reddit and Facebook groups. Qualtrics facilitates the measurement of response times, with the primary time metric being the total number of seconds the question was displayed before the participant submitted their final response.

A total of 362 developers participated in the experiment (see Table 2). As the experiment deals with the most basic constructs in programming, it was felt that all developers are valid participants and there is no need for screening. Among those who reported their educational background, 48% held a Bachelor's degree or were studying for one, 25% had or studied for a Master's degree, and 6% a PhD. The others were either self-taught, had vocational training, or learned programming in high school. Regarding professional experience, 13% reported having 0-2 years of experience, 30% had 2-6 years, and 57% had more

than 6 years of experience. In terms of gender, among those who disclosed this information, the participant group was predominantly male, comprising 95% men and 5% women. While extreme, this is close to the ratio in the Stack Overflow developer survey[4].

5 Results

For each code snippet we have two results: the fraction of participants who understood it correctly, and the times it took them to do so. In the figures below, for instance Figure 3, we show the CDF (cumulative distribution function) of the times. The time is on the horizontal axis, and the graph shows the fraction of participants who solved the problem correctly in up to a certain time. Thus a line that is more to the right reflects the need for more time to give a correct answer. We represent incorrect results by assigning them a time of infinity. As a result the CDFs converge to the fraction of correct answers and not to 1. In other words, the fraction of incorrect answers can be read off the right end of the graphs as the gap from the top. The horizontal axes are trimmed where such convergence is achieved; however, in the experiment itself participants were allowed as much time as they wanted.

In addition, Table 3 presents a summary of all the results, showing the median time to achieve a correct answer and the percentage of correct answers for all the individual expressions and for several groupings of expressions. It also contains the p-values of comparisons between pairs of equivalent expressions, as further discussed below. As multiple comparisons were conducted, we employ Bonferroni correction, and mark the results that remain statistically significant after such a correction.

5.1 Level of Comprehension

First, we compare between the cumulative results in the tracing part and in the comprehension part, which is the subject of RQ1. In Figure 3 and the bottom of Table 3, we can observe that the differences are highly significant: tracing expressions is substantially faster (the CDF rises much more sharply) and leads to much fewer errors (it converges to a much higher value) than comprehension of the expressions. This considerable gap clearly indicates a fundamental difference between the two levels of understanding. Tracing appears to demand less cognitive effort and is significantly easier. And this is also true for each expression individually, as indicated by comparing the left and right sides of each row in Table 3.

Formally, the independent variable has two levels, representing the level of understanding. The dependent variable is the time of responses. The comparison is between subjects, as explained above. We would like to check whether the average time needed to process the different levels is equal. For this we

[4] The question about gender was last included in 2022, and 92% identified as male.

Table 3 Summary of results for all expressions. Comparisons in the top part are between pairs of equivalent expressions (De Morgan). The last comparison is between tracing and comprehension.

	tracing			comprehension		
expression	time	good	p-value	time	good	p-value
$\neg p \vee \neg q$	9.47	91%	0.806	21.61	55%	0.001*
$\neg(p \wedge q)$	8.38	88%		13.68	68%	
$\neg p \wedge \neg q$	9.25	96%	0.005	15.76	95%	0.161
$\neg(p \vee q)$	8.68	88%		13.68	85%	
$p \wedge \neg q$	6.39	96%	$< 10^{-15}$*	10.77	92%	$< 10^{-15}$*
$\neg(\neg p \vee q)$	11.85	85%		22.14	60%	
$\neg p \wedge q$	7.01	96%	$< 10^{-15}$*	12.21	89%	$< 10^{-15}$*
$\neg(p \vee \neg q)$	12.22	86%		19.55	68%	
$p \vee \neg q$	7.33	98%	$< 10^{-15}$*	16.94	56%	0.035
$\neg(\neg p \wedge q)$	11.85	85%		22.89	51%	
$\neg p \vee q$	7.19	93%	$< 10^{-15}$*	20.94	52%	0.839
$\neg(p \wedge \neg q)$	12.59	85%		20.05	61%	
$p \vee q$	4.18	98%	$< 10^{-15}$*	11.74	81%	$< 10^{-15}$*
$\neg(\neg p \wedge \neg q)$	13.90	78%		20.16	59%	
$p \wedge q$	4.44	97%	$< 10^{-15}$*	7.42	95%	$< 10^{-15}$*
$\neg(\neg p \vee \neg q)$	13.89	79%		22.36	52%	
all AND	8.54	91%	0.208	13.98	76%	10^{-11}*
all OR	8.99	90%		17.56	63%	
all total	8.81	90%		15.76	70%	$< 10^{-15}$*

time: median of times of correct answers, in seconds
good: percent of answers that were correct
* remains statistically significant after Bonferroni correction

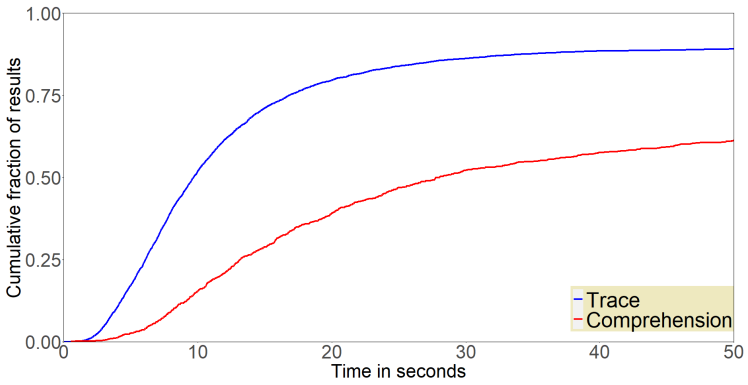

Fig. 3 CDFs of the time to correct answers for all logical expressions for the trace questions (part 1) compared to the comprehension questions (part 2).

used a t-test, where the null hypothesis is that the times are equal, and the alternative hypothesis is that the expected values are different. The results of this test is that there is a statistically significant difference (p-value $<$ 2.2e-16). The test is performed on the data as presented in the graph of Figure 3, with the times truncated at 50 seconds, and wrong answers taken as this maximal value.

A possible internal threat to the validity of this conclusion is that the difference may be due to the introduction of the Venn diagram used to assess the comprehension of expressions in part 2 of the experiment. Maybe participants took more time to read the diagram, and made more errors in reading the diagram, not in figuring out which shapes lead the Boolean expressions to evaluate to TRUE. We believe that this is not the case for two reasons.

First, dual coding theory highlights an important difference between comprehending Boolean expressions for general inputs and their comprehension for a specific input. This difference is the level of concreteness [47,46,14].

Visualizing a yellow triangle, as in part 1 of the experiment, is concrete. In the Venn diagram such a concrete visualization exists only for the intersection (area 3). Areas 1 and 2 are partly concrete, representing triangles of any color and any shape that is yellow, respectively. But area 4, the outside area, lacks any concreteness: a shape that is neither a triangle not yellow can be a red square, a blue circle, or a purple star. Thus it is not possible to visualize what this area represents.

According to dual coding theory, the cognitive processing of abstract language that does not have a concrete visualization is approximately twice as hard as processing concrete language, because the non-verbal system cannot aid the processing of such abstract inputs [46, p. 900]. However, needing to consider the non-concrete options of area 4 in the Venn diagram is an integral part of the question. We therefore contend that this difficulty is inherent in the task, and not a result of introducing a visual representation in the form of a Venn diagram. And indeed, our results agree with the predictions of the dual coding theory.

Second, the results below in Section 5.2 show a difference between expressions with different logical operators. As these results were obtained using exactly the same methodology with the Venn diagram, we can conclude that the semantics of the Boolean expression at least had a larger effect than the introduction of the diagram. In other words, using the diagram did not mask real differences between the comprehension of different Boolean expressions.

Given that tracing appears to be easier, we also looked for a simple model that can explain the distribution of results for the different expressions. Specifically, we checked a model of how the median time for achieving a correct answer depends on the number of negations and the structure of the expression. The data from Table 3 is shown in Figure 4. As can be seen, for tracing good simple models are indeed found. When the expression is simple, the model for time is $t = 4.4 + 2.5n$, where n is the number of negations. When the expression has a negation applied to a sub-expression, the model is $t = 6.3 + 2.7n$. These models are excellent, with R^2 of 0.98 and 0.93 respectively. We can therefore say that adding a negation "costs" approximately 2.6 seconds more, and negating the whole expression "costs" an additional 2 seconds. But when we look at the data for semantic comprehension, we see that the data points do not line up on straight lines, and indeed, the trend lines are a poor fit, with R^2 values of 0.61 and 0.47. Likewise, models for the percentage of correct answers as a function of negations are relatively poor.

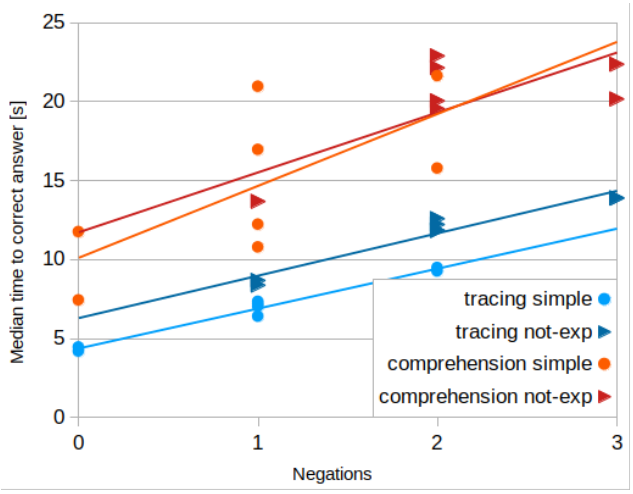

Fig. 4 Linear models of time to correct answer as a function of the number of negations and the expression structure. Simple: a simple condition of two literals. Not-exp: negation applied to a simple expression.

Our conclusions from all these results are twofold. First, developers' understanding during code tracing differs from fully comprehending it. Tracing is on average determined by the syntactic structure of the expression, and specifically the negations in contains. Comprehension is considerably more challenging, as indicated by the increased time it requires and the higher number of errors. It is also subject to additional factors, as witnessed by the poor fit of models based only on negations. Second, the metrics of time and correctness do not measure exactly the same thing. For simple expressions, negations well predict the time needed to trace them, but not the errors. A difference between the time and correctness metrics has been noted before (e.g. by Ajami et al. [2]), but analyzing it is beyond the scope of this paper.

5.2 Logical Operators

In this section, we examine the influence of the logical operators AND and OR on the understanding of logical expressions, in both the tracing and the comprehension levels, in order to answer RQ2. For the comprehension questions we note that summing the number of regions in diagrams for all OR questions and for all AND questions yields equality, indicating no bias in the number of clicks needed to answer. Figure 5 shows the results.

This analysis provides perhaps the most surprising and intriguing result of the whole experiment. We can observe that while there is no significant difference between the operators in terms of tracing (graphs are essentially identical and p-value $= 0.208$), there is a notable gap between them at the comprehension level, with the AND operator being significantly easier to com-

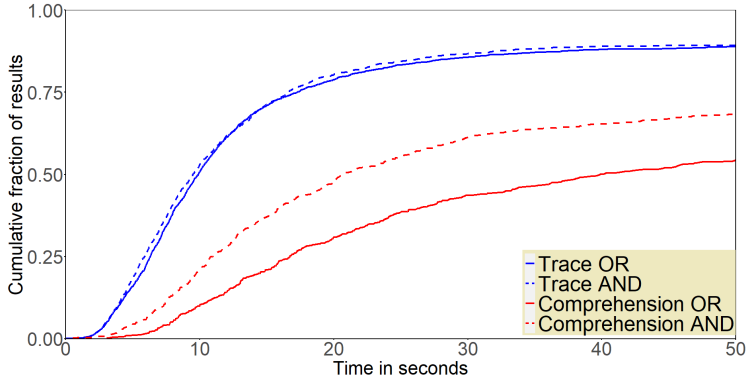

Fig. 5 CDFs of the time to correct answers for AND and OR operators in both tracing and comprehension levels.

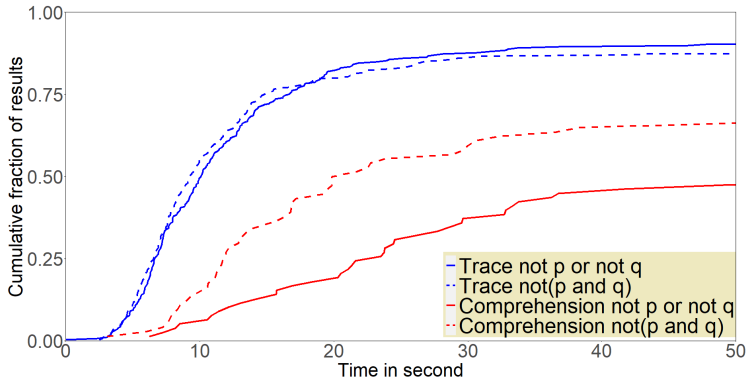

Fig. 6 CDFs of the time to correct answers for the equivalent logical expressions $\neg(p \wedge q)$ and $\neg p \vee \neg q$, in both tracing and comprehension levels.

prehend (both faster and fewer errors, p-value = 1.04e-11). This suggests that while OR is on average not significantly more challenging syntactically during tracing, it does significantly increase cognitive effort for comprehension. This testifies that some interesting cognitive process is at work. We discuss this further below, in Section 6.

Note, however, that the equivalence of the operators under tracing is "on average". As we will see below, in certain questions a difference between the operators was in fact observed, but when taken together these differences cancel out.

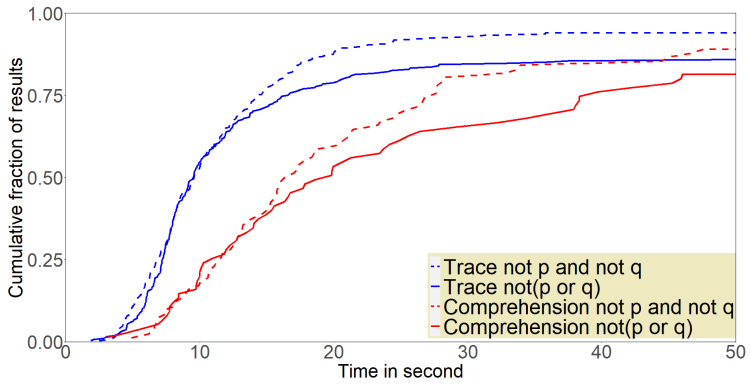

Fig. 7 CDFs of the time to correct answers for the equivalent logical expressions $\neg(p \vee q)$ and $\neg p \wedge \neg q$, in both tracing and comprehension levels.

5.3 Equivalent Expressions

In this section, dealing with RQ3, we compare the difficulty of understanding equivalent Boolean expressions, based on De Morgan's laws. A summary of all the results was given above in Table 3. As before, two levels of understanding are considered: tracing and comprehension. First, we will examine the following expressions:

- De Morgan's first pair: $\neg(p \wedge q)$ and the logically equivalent $\neg p \vee \neg q$
- De Morgan's second pair: $\neg(p \vee q)$ and the logically equivalent $\neg p \wedge \neg q$

In Figures 6 and 7, we can observe the results for the aforementioned pairs in each of the understanding levels. Starting with the tracing level, it is evident that for the first pair there is no difference (p-value = 0.806), whereas for the second pair there is a significant albeit slight difference (p-value = 0.005). However, at the comprehension level, there are differences in both pairs—but for the second pair, it is only for part of the distribution and therefore not statistically significant (p-value = 0.001, p-value = 0.161). Interestingly, the pattern is reversed between the pairs: in the first pair the expression with a NOT applied to a sub-expression is much more readable, whereas in the second pair, the NOT applied to a sub-expression is less readable. But this aligns with the previous finding that the OR operator is more complex for comprehension: in the first pair, expression $\neg(p \wedge q)$ is more readable then $\neg p \vee \neg q$, while in the second pair, expression $\neg p \wedge \neg q$ is more readable then $\neg(p \vee q)$.

An additional observation is that the first pair of expressions is less readable than the second pair, both in terms of time for correct understanding and even more so in terms of errors made. But note that these are not directly comparable, as the expressions are not logically equivalent.

We also conducted comparisons between equivalent Boolean expressions that contain more negations. We will present only two of these, as the other patterns are similar. In Figure 8, the result for expressions $p \vee \neg q$ and $\neg(\neg p \wedge q)$

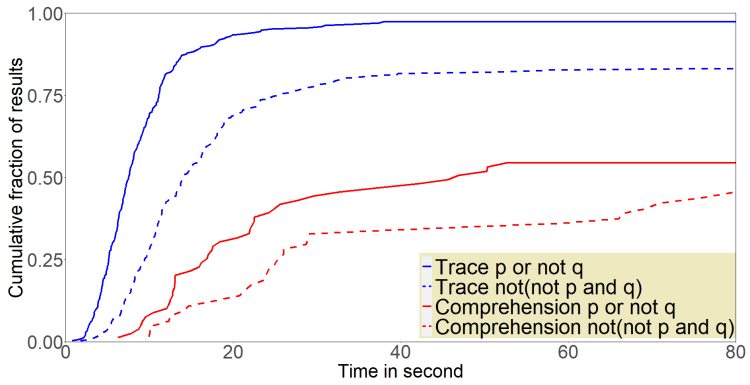

Fig. 8 CDFs of the time to correct answers for the equivalent logical expressions $p \vee \neg q$ and $\neg(\neg p \wedge q)$, in both tracing and comprehension levels.

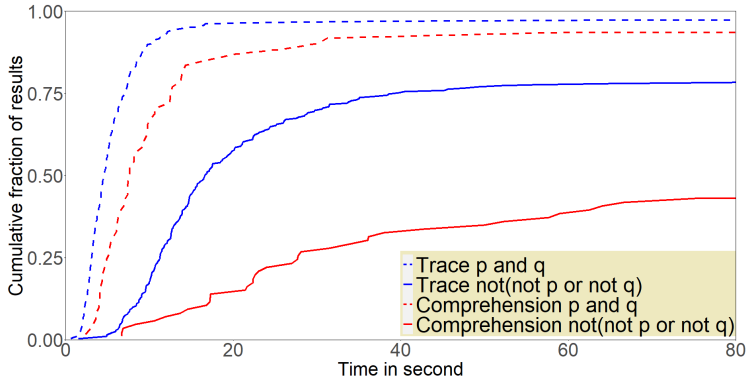

Fig. 9 CDFs of the time to correct answers for the equivalent logical expressions $p \wedge q$ and $\neg(\neg p \vee \neg q)$, in both tracing and comprehension levels.

is displayed. It is evident that the expression with both external and internal negations (a total of two negations) is more complex in both levels of understanding: both in tracing and in comprehension (p-value $<$ 2.2e-16, p-value $=$ 0.035). In fact, it seems that having multiple negations, and especially applying a negation to a sub-expressions that includes a negation, has a larger effect than the logical operator. The negations cause more errors to be made in the tracing level, and also at the comprehension level having the AND operator does not fully compensate for this.

In Figure 9, we see the result for the expressions $p \wedge q$ and $\neg(\neg p \vee \neg q)$. There is a wide gap in both levels of comprehension between the expression with multiple negations and that without any negation (both p-values $<$ 2.2e-16). This wide gap is the result of the confluence of all three factors:

- The difference between AND and OR
- The difference between multiple negations (3 of them) and no negations

– The difference between a complex expression with a negation of a sub-expression with negations and a simple expression with no such structure

So in the first pair there is a big difference between understanding levels and a smaller difference between the pair of expressions, and in the second there is a small difference between the levels and a larger effect between versions, because both the effect of the operator and the effect of negations coincide.

5.4 Short-Circuits

In the tracing part of the experiment, we covered all possible assignments for conditions involving two literals—whether the first had a value of TRUE or FALSE, and whether the second had a value of TRUE or FALSE. Since our conditions involve only two literals, each expression resulted in four variations. Some of these assignments allowed for the possibility of reading shortcuts, where the truth value of the condition could be determined based on just one literal, given the logical context of the expression. In order to examine RQ4 we compared the results of cases where shortcuts could be used or not, and how significant this was.

Interestingly, we found that using short-circuits also depends on the logical operator (Figures 10 and 11). When the operator is AND, there is absolutely no difference (p-value = 0.208) between the tracing of expressions that allow short-circuits and those that do not. But for expressions with OR there is a small but statistically significant difference between expressions that allowed short-circuits and those that did not (p-value < 0.001, effect size of 0.147, which is considered negligible). Our conclusion is that there is a tendency to short-circuit, but only for conditions with OR, and that with only two literals, the effect size is not really meaningful.

Note that using short-circuits led to shorter response times and also to fewer errors. Thus our experimental participants did not make mistakes as a result of performing the short-circuits, as may happen with students [55]. However, in our expressions the order of evaluation was immaterial, reducing the opportunity to err.

The reason OR operators are more frequently subject to short-circuits may lie in the fact that a short-circuit with OR occurs when the first literal evaluates to TRUE, thereby making the entire expression TRUE. In contrast, for short-circuits involving the AND operator, the short-circuit is applied when the first literal evaluates to FALSE, which makes the entire condition FALSE. This aligns with previous research suggesting a cognitive bias towards understanding expressions whose truth value is TRUE [4]. This phenomenon might have an effect on the tendency to apply short-circuits during evaluation.

The fact that developers sometimes take shortcuts is interesting, even with the limited impact observed here, for two main reasons. First, in real-world conditions, a logical condition may contain more than two literals, which could make the impact of short-circuits more pronounced. Additionally, in this study,

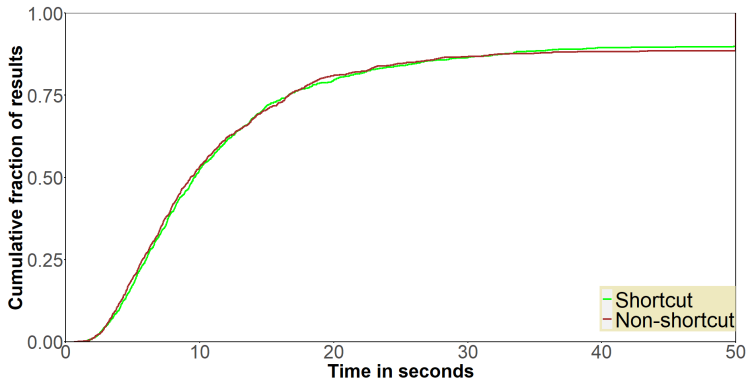

Fig. 10 CDFs of the time for all logical expressions with AND with possibility to short-circuit compared to those without this option.

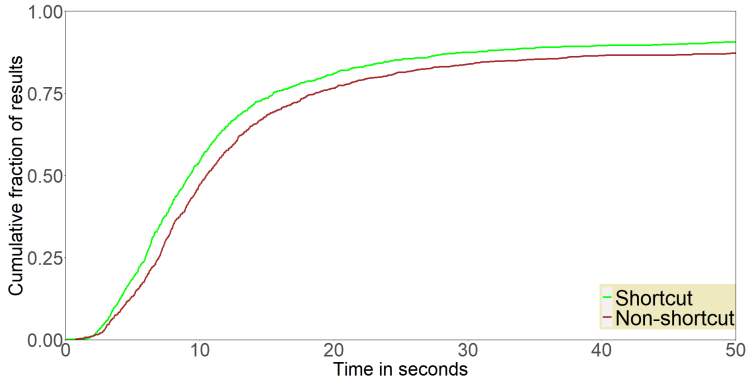

Fig. 11 CDFs of the time for all logical expressions with OR with possibility to short-circuit compared to those without this option.

the literals were designed to be easy to process (for necessary methodological reasons), but in actual code, a second literal might be more complex to interpret, potentially making the short-circuit more significant.

6 Theories of Common Mistakes

In several of the previous results we observed that many mistakes were made by participants in the experiment, especially in the comprehension part with the Venn diagram. There were more mistakes in expressions involving the OR operator relative to those with AND (Section 5.2). But the mistakes were not only in expressions with OR, and importantly, they were not random: some mistakes were much more common than others, and were made much more frequently (e.g. Figure 13 below). In an attempt to understand these results, we analyzed the common mistakes observed for different expressions. This allows

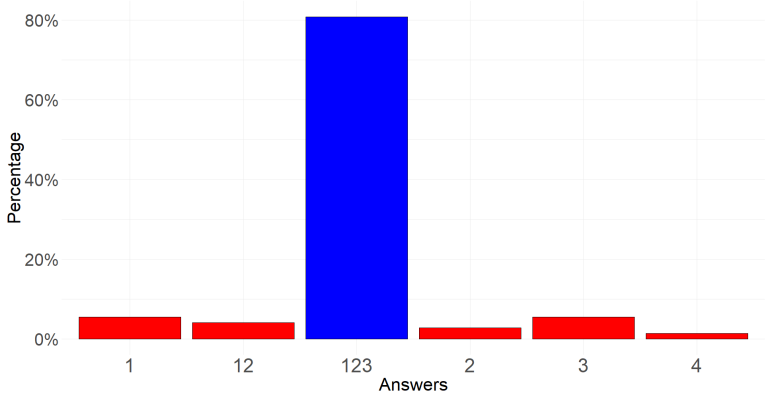

Fig. 12 Responses to comprehension of $p \vee q$. Numbers refer to the areas in the Venn diagram of Figure 2. The correct answer is in blue, and mistakes in red. In this case, no specific mistakes stand out. Combinations that did not appear in the experimental results (e.g. 13, 14, 23, 134, etc.) were excluded from the graph.

us to suggest theories of what causes confusion, and of the cognitive processes that influence the comprehension of logical conditions in code. In particular, it reveals a pattern of interactions between the NOT and OR logical operators at the comprehension level of understanding.

The literature identifies two kinds of errors in processing Boolean conditions in code. The first is the tendency to confuse the logical OR operator with the logical XOR operator [28,26]. In natural language, "or" is often interpreted with the meaning of "either or", implying that one or the other of the clauses is true, but not both. In logic terms this means to understand the OR operator as if it was a XOR, leading novice developers, who are less accustomed to formal logical thinking, to misinterpret the semantics of different logical expressions.

The other error discussed in the literature is the confusion between the logical AND and OR operators [21]. But saying that programmers confuse the AND and OR operators only describes the error, and does not explain *why* this phenomenon occurs, or the cognitive processes that lead to it. Moreover, it does not clarify why this confusion arises only in certain logical expressions while being absent in others.

To address this gap, we introduce alternative cognitive theories that provide a more comprehensive account of common logical errors. These theories will not only explain additional mistakes that are not accounted for in the existing literature, but will also clarify why these errors occur specifically in certain expressions and not in others. By doing so, they will offer what we believe to be a more precise and complete explanation of these logical misconceptions. In all the following, we focus on the comprehension questions using the Venn diagram.

Given that there were more mistakes in expressions with OR, we start with these expressions. The simplest one is $p \vee q$. The distribution of answers given

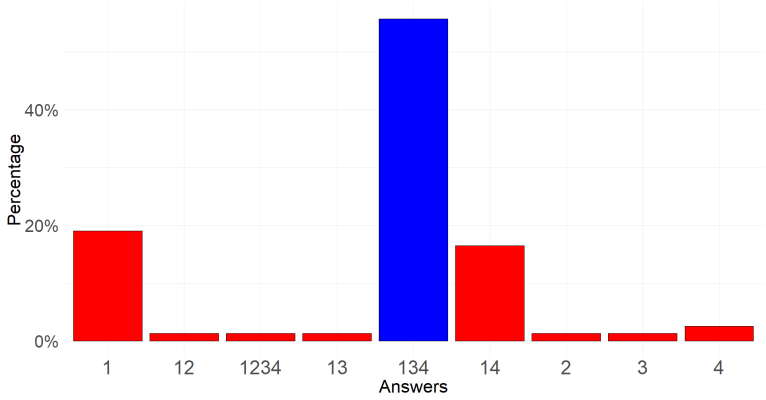

Fig. 13 Responses to comprehension of $p \vee \neg q$.

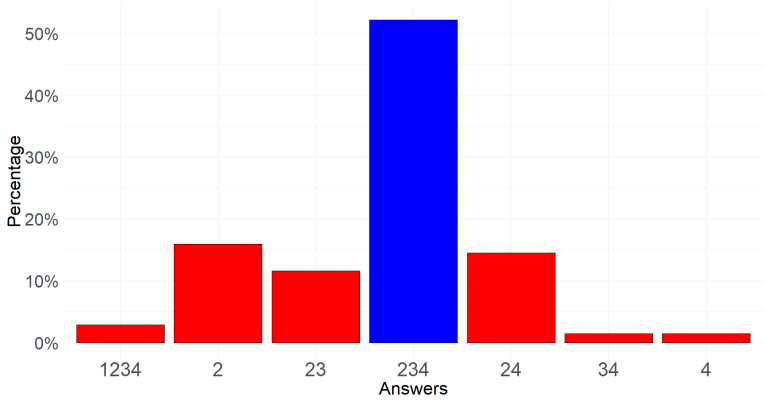

Fig. 14 Responses to comprehension of $\neg p \vee q$.

by participants in the experiment is presented in Figure 12. The labels on the bars in the figure indicate the different regions in the Venn diagram of Figure 2: 1 represents the region of "pure p", 2 represents the region of "pure q", 3 represents the intersection of p and q, and 4 represents the external region, which includes elements that are neither in p nor q. The correct answer is 123, meaning the union of p and q (that is, the areas of pure p, pure q, and the intersection). As can be seen, there are relatively few mistakes, and there is no single mistake that stands out.

Figure 13 presents the distribution of responses for the logical condition $p \vee \neg q$. Importantly, in this case we see that mistakes are not random: there are just two common patterns of mistakes, and the others are quite rare. The correct answer is areas 1, 3, and 4, that is all areas except the "pure q" area 2. The common mistakes are subsets of these areas: either 1 alone (the "pure p" area), or 1 and 4 (the "pure p" and the external areas). In both cases, area 3, the intersection of p and q, is missing.

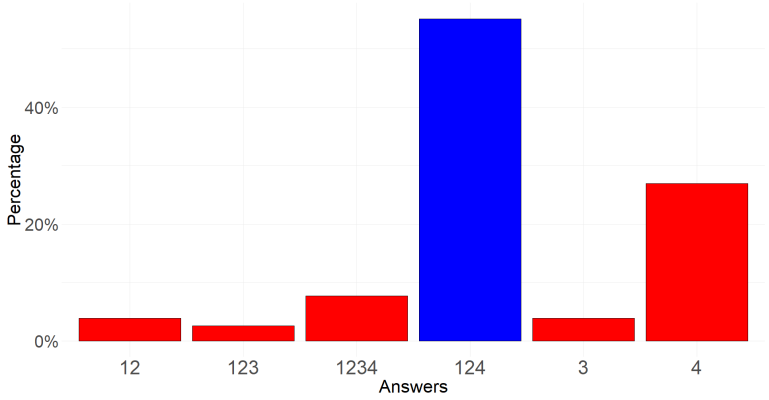

Fig. 15 Responses to comprehension of $\neg p \vee \neg q$.

A similar phenomenon is observed with the mirror expression $\neg p \vee q$, in Figure 14. Here too the most common errors involve marking q without the intersection with p, namely area 2, either with or without the external region 4. (In addition, a slightly less common mistake is to mark areas 2 and 3; we ignore this for the moment).

Finally, the last simple expression with OR is $\neg p \vee \neg q$, where both p and q are negated. In Figure 15 it can be observed that the most common mistake for this expression was to mark only the external region 4, outside of p and q.

Comparing Figure 12 with the other three, we can conclude that the difficulty in comprehending these expressions is not due to the OR operator itself, but to the interaction of OR with negation. This suggests that some cognitive difficulty is involved when these logical operators are combined. Our goal is to find the source of this difficulty.

We start with a simple example. Recall that we focus on cases involving two literals, connected by the logical OR operator, where at least one of these literals is a variable negated by the NOT operator. An example is the expression "square OR NOT green". This expression evaluates to TRUE for various shapes, including green squares. However, a green square directly contradicts the negated literal "NOT green". We suggest that this creates a cognitive dissonance, which manifests itself in a difficulty to include it among the elements for which the formula is satisfied.

More generally, in expressions with the AND operator the literals have a cumulative effect. Objects that satisfy the expression also satisfy each of the literals. But in expression with the OR operator this is not the case. For such expressions, an object can directly contradict one of the literals, but still satisfy the expression as a whole. We theorize that when a logical expression is satisfied for objects that directly contradict a negated variable, there is a cognitive difficulty to incorporate these objects into the final answer. This theory explains why the intersection of p and q is sometimes left out in the

Table 4 Common mistakes and their possible explanations. Answers and mistakes are noted by their areas in the Venn diagram of Figure 2: 1 represents the region of "pure p", 2 represents the region of "pure q", 3 represents the intersection of p and q, and 4 represents the external region, i.e. neither in p nor in q. Common mistakes are those that occurred in at least 8% of the answers. "–" means the explanation is inapplicable in this case.

			Explained by						remain
Expression	correct answer	common mistakes	OR as XOR	mix OR AND	contradict NOT	bad De Morgan	forget NOT()	forget area 4	unexplained
$p \wedge q$	3		–		–	–	–	–	
$\neg p \wedge q$	2		–			–	–	–	
$p \wedge \neg q$	1		–			–	–	–	
$\neg p \wedge \neg q$	4		–			–	–		
$p \vee q$	123				–	–	–	–	
$\neg p \vee q$	234	2, 23, 24		2	2, 24	–	–	2, 23	
$p \vee \neg q$	134	1, 14		1	1, 14	–	–	1	
$\neg p \vee \neg q$	124	4		4	4	–	–		
$\neg(p \wedge q)$	124	4	–	4	–	4			
$\neg(\neg p \wedge q)$	134	1, 14, 3	–	1		1		1	14, 3
$\neg(p \wedge \neg q)$	234	2	–	2		2		2	
$\neg(\neg p \wedge \neg q)$	123	4, 3, 1	–	3		3	4	–	1
$\neg(p \vee q)$	4				–				
$\neg(\neg p \vee q)$	1	14						–	14
$\neg(p \vee \neg q)$	2	4						–	4
$\neg(\neg p \vee \neg q)$	3	4, 123		4, 123	4, 123	123	4	–	

expressions $p \vee \neg q$ and $\neg p \vee q$, and why the "pure p" and "pure q" areas are left out for $\neg p \vee \neg q$.

Additional common errors can be explained by other theoretical perspectives. Many errors occur in compound expressions, where an external negation operator is applied to a complete internal expression (e.g. $\neg(p \wedge q)$). In these cases novices may incorrectly apply De Morgan's laws, by distributing the negation without properly inverting the internal logical operator. This phenomenon accounts for frequent errors in such expressions. Notably, this explanation serves as a specific instance of the previously noted error of swapping the AND and OR operators. However, it provides a more precise account of why this mistake occurs particularly in expressions of this form.

Another class of errors is attentional errors, where certain elements of the expression or the answer are mistakenly overlooked. One common attentional error, in compound expressions with an external NOT operator, involves overlooking the effect of this external negation altogether. When this happens the given answer is the correct answer for the internal expression alone — or a common error made for the internal expression alone. Both these options appear to have happened in the most complicated equations we used, $\neg(\neg p \wedge \neg q)$ and $\neg(\neg p \vee \neg q)$, respectively.

Another attentional error occurs when the expression is satisfied in the external region in a Venn diagram, i.e., elements that do not belong to either variable p or q, but these elements are mistakenly overlooked. This error was common in expressions where one variable was negated and the other was not.

In Table 4 we present all the logical expressions we studied, with all the common mistakes made in them, and what theory explains each one. Common mistakes are defined to be those that occurred in at least 8% of the answers in the experiment. This threshold was chosen in an attempt to balance between capturing cases where some answers appear more common than others but are quite low in absolute terms, and cases where many answers all appear at a similar rate, and the top one may be quite high but not really distinguished from the others.

Surprisingly, the OR as XOR theory from the literature does not account for any of the common errors identified in our experiment. This may be due to the fact that the developers who participated in the experiment are not early-stage computer science students. Consequently, they may have already developed a more refined understanding of logical operators, reducing their susceptibility to this specific type of confusion.

The other thesis from the literature, the substitution of AND with OR, apparently does explain many of the common errors. However we note that all these cases can also be accounted for by the theories we propose in this paper. We argue that the substitution of AND with OR is sometimes merely a description of a symptom rather than a real explanation of why these specific cases lead to such confusion. Our proposed explanations clarify *why* the tendency to swap logical operators arises in particular cases. For example, the common errors in the four expressions where a NOT is applied to an expression with AND can be explained by distributing the NOT, but failing to change the AND to OR as required by De Morgan's laws.

Furthermore, our explanation concerning contradictions between the final truth value and a negated literal accounts for additional errors that cannot be explained solely by the theory of operator substitution. specifically, in the logical expressions $\neg p \vee q$ and $p \vee \neg q$, the observed common errors 24 and 14 remain unexplained with the thesis of mixing AND and OR. However, they can be explained with the thesis of a contradicted NOT we proposed. In both of these cases, the intersection (area 3) is missing from the answer. The reason for this is that, in the first expression the intersection contradicts the literal $\neg p$, and in the second it contradicts the literal $\neg q$.

The expression $\neg p \vee \neg q$ suggests an interesting observation about the theories. In this case both explanatory approaches align, since the common errors fit both interpretations, but we suggest that this alignment may be coincidental. This is because the AND-OR explanation does not clarify why a similar substitution does not occur in the parallel expression $\neg p \wedge \neg q$. In contrast, our contradiction-based theory provides a clear rationale for why the phenomenon appears specifically in $\neg p \vee \neg q$. Similarly, in other expressions involving the AND operator, there is also no observed tendency to swap logical operators; the substitution explanation does not account for this asymmetry, whereas our proposed theory does.

Additional common error patterns are related to the presence of an external negation. These errors can be classified into three main types:

- The internal expression was evaluated correctly, but the external NOT was not applied to this result; the common error is therefore the answer for the internal expression.
- The internal expression was evaluated erroneously, *and* the external NOT was not applied; the common error observed is therefore simply the common error of the internal expression.
- The internal expression was evaluated erroneously, and then the external NOT was applied to this error; The common error is therefore the complement of the common error that occurs in the internal expression.

An example of this can be observed for the expression $\neg(\neg p \vee \neg q)$. As can be observed, one of the common errors that occurred is 4, which is the same as for the internal expression $\neg p \vee \neg q$. And the other common error is its complement, 123, which may be obtained by applying the external NOT to the first error.

Note that the common error 4 results from an interaction between two factors: an incorrect computation of the inner expression, following the same types of errors previously discussed, and the failure to apply the external NOT. This suggests an interplay of cognitive processes: first, the inner expression is miscalculated, and second, the external NOT is overlooked, leading to a compounding effect that produces the observed mistake.

7 Implications

We now turn to the implications of our findings.

7.1 Writing Readable Expressions in Code

The main conclusions regarding code writing are divided into two parts. The first conclusion is that code with multiple negations makes understanding more difficult at both levels, tracing and comprehension. This replicates previous results, e.g.[4]. However, it is not just a matter of counting negations. Negating a whole expression is different from negating a single variable, and negating all the variables may be more readable than negating just a subset (an effect called "lexical regularity" in Baron et al. [4]). The second significant factor is the logical connection between the literals. We demonstrated that the OR operator is more complex to comprehend compared to the AND operator, although at the tracing level there is no difference between them. This is due to an interaction between OR and NOT.

The conclusion is that the way a condition is expressed may have a significant effect on how easy it is to understand correctly. Therefore, when writing any logical condition it may be beneficial to consider its different formulations (which are easily generated using De Morgan's laws). Then, given the logically equivalent expressions, those with fewer negations are usually preferable, and double negations (e.g. having a negation that is applied to a sub-expression

that also includes a negation) should be avoided. Furthermore, expressions using the AND operator are also preferable.

For example, when using De Morgan's laws to generate alternatives to simple conditions, we would prefer the expression $\neg(p \wedge q)$ over over the equivalent expression $\neg p \vee \neg q$. At the same time, we would prefer expression $\neg p \wedge \neg q$ over its equivalent expression $\neg(p \vee q)$, even though the more readable conditions may appear different. This is due to the logical operator and the semantic comprehension it demands.

7.2 Methodology of Code Comprehension Experiments

Code tracing is a common task used in program comprehension experiments. However, such experiments should consider tracing only as a partial approach, as it reflects a more superficial understanding of the code. It is crucial to acknowledge the limitations of tracing-based experiments. Specifically, in the context of logical conditions in code, tracing can be performed using short-circuits applied to logical conditions. As we have seen, there is some tendency to make these short-circuits. Therefore, experiments based on tracing should account for the possibility of short-circuits in conclusion-drawing, and also for the inherent limitations of the level of understanding they evaluate.

It is important to emphasize that code comprehension involves multiple levels, each more abstract than the other. Experiments should always define the specific level being assessed. After the tracing level we have comprehension, which pertains to understanding the behavior of the code for all possible inputs. Above that is a more abstract level, which is the functional-level understanding, describing the purpose of the code and what it is intended to achieve in a wider context. For example, in the case of calculating a purchase price, one can trace the code and understand its output for a specific input. The next level is to generalize, which involves understanding the output for all possible inputs, for example that above a certain threshold you add X% to the price. Beyond this lies the semantic level, where there is a broader understanding of the *purpose* of the code—in our example, perhaps this is the implementation of a local tax regulation. However, when considering simple expressions in isolation, as we do here, such a broader purpose does not exist.

When designing code comprehension experiments, it is essential to recognize that the difficulty measured at a particular level may not necessarily reflect the complexity and differences at higher levels of understanding. Many code comprehension experiments are limited to the tracing level, perhaps because this level ("what does the following code print?") is the easiest to implement and check automatically. But in many cases it is the higher levels that are important. For example, when a developer needs to review code generated by an AI tool, it is not enough to verify that the result is correct for a couple of specific inputs. It is crucial to verify that it is correct for *all* inputs, and that it conforms to the domain-level specification. In other words, the requirement

is to achieve semantic understanding—and this is measured only by semantic-level experiments.:

8 Threats To Validity

Construct validity: How to measure the difficulty of understanding is subject to debate. In our experiment this was operationalized by the time needed to produce a correct answer and by the fraction of wrong answers. Measuring both the time and correctness of responses is a common practice [45], and in all our results they were consistent: treatments that took more time also suffered from more errors. However, while these metrics are a common proxy for difficulty of understanding, they are not the same as difficulty of understanding. But in our analysis the measured times are not important in absolute terms, but only relative to the times measured for other expressions. The results therefore can indeed give a perspective on the relative hardness of different expressions.

Internal validity: Several decisions concerning the design of the experiment were taken to improve internal validity. Because tracing reflects a lower-level of understanding, placing comprehension first could have altered reading and understanding patterns during the tracing phase, as achieving comprehension could certainly aid in tasks relevant to tracing. To prevent this, we structured the experiment so that the comprehension part was always last, ensuring a clean environment to assess tracing. Although tracing does not impact comprehension (as semantic understanding cannot be achieved solely through tracing), the consistent placement of comprehension tasks at the end could introduce bias when comparing results across the levels (for RQ1). This is particularly relevant because the comprehension part contains questions requiring multiple clicks to answer, whereas each tracing question required only one click (compare Figure 1 and 2). In addition, participants might experience fatigue by the time they reach the comprehension part.

Nonetheless, we believe these differences are minimal, as the performance gap between the sections was substantial and significant, even though comprehension tasks might benefit from the logical familiarity developed in the tracing stage. Regarding the click count, the stark differences in time and errors suggest that the discrepancy is not solely due to the number of clicks. Moreover, in all comparisons of conditions within the comprehension section, click requirements were consistent across tasks. For example, in RQ2, the total number of clicks for all AND conditions equaled those for all OR conditions. Similarly, when comparing the comprehension of logically equivalent expressions (RQ3), these pairs by definition required the same number of clicks as they led to identical answers, thus eliminating any click-based bias in our comparisons. As for the effect of fatigue, previous work indicates that its effect is more on attrition, where participants may decide not to complete the experiment, than on measured results [2].

Lastly, encountering the first question in a section might lead to some initial confusion as participants adapt to the task, potentially skewing results compared to subsequent questions [2]. However, each section included a preliminary explanation to introduce the format, structure, and expectations for that part. Additionally, question order was randomized for each participant, so any sequence-related bias, if present, did not systematically favor (or harm) any specific question. Consequently, our results remain unbiased when comparing questions within the same section.

External validity: Research findings are always limited to the circumstances under which they were derived. There are a lot of possible structures of logical expressions. Our research examined only a limited number of short, basic formulas, isolated from any surrounding code. This was done as an exploration of the possible effects of the atomic elements of Boolean expressions and their basic interactions. But it is important to acknowledge that the results for the expressions we employed may not necessarily generalize to other scenarios or expressions. There is no alternative to performing additional experiments, with increasingly complex expressions, to get a fuller picture.

Another aspect of generalizability concerns the developer population. The participants in our study were predominantly male, this may raise concerns about the generalizability of the findings to female developers. However, we note that developers are indeed predominantly male (as witnessed, for example, in the Stack Overflow developer survey). So our results are indeed representative for the vast majority of developers in this respect.

9 Conclusions

Understanding Boolean conditions is a significant component in grasping the logic and functionality of code. However, understanding occurs at varying levels, which we believe this research effectively demonstrates. Our experiment focused on two levels of understanding Boolean conditions: the *tracing level*, which involves processing the condition's value for a specific input, and the higher *comprehension level*, which we defined as understanding the condition's meaning by identifying all inputs for which the condition holds TRUE or FALSE. To achieve this, we employed questions specifically tailored to assess understanding at these distinct levels.

Our findings reveal substantial differences between tracing and comprehension in the interpretation of Boolean expressions in code. In addition, a significant interaction emerges between the level of understanding and the operator used in the expression. While on average there is no notable difference in understanding the AND versus OR operators at the tracing level, at the deeper level of comprehension, OR expressions pose a greater cognitive challenge than AND expressions. This highlights the importance of differentiating between levels of understanding in code.

We believe that this experiment has broader implications for code comprehension studies. Typically, code comprehension experiments focus on a single level of understanding, or they lack a clear definition of the required level. We believe it is essential to specify which level of understanding is being tested in an experiment, not least so because one level of understanding does not necessarily translate to another. For example, our experiment demonstrated cases where results at the tracing level differed from those at the comprehension level for the same parameter. Furthermore, although many code comprehension studies emphasize the tracing level, we contend that examining deeper levels of code understanding is increasingly important. This is particularly significant in contemporary programming practices, where developers frequently review code generated by AI systems. In these contexts, it is often crucial to thoroughly understand the implementation, beyond merely tracing specific inputs.

Understanding Boolean conditions, and specifically also those involving negation, is integral to the world of code. The comprehension of statements with negation has long been a prominent focus in cognitive research on natural language processing, producing a substantial body of work [17,29]. Our work extends this line of research into the domain of code—a complex realm that combines elements of natural language with those of formal language. We believe this expansion can lead to mutual enrichment of both fields. On the one hand, it may broaden cognitive research on natural language by introducing an entirely different context, potentially yielding new insights. On the other, it could illuminate best practices for writing more readable code and enhance our understanding of the cognitive processes developers engage in as they interpret code segments containing certain Boolean conditions.

Obtaining a fuller picture would however require much additional work. Our study was limited in the sense that we only considered the effect of varying the number of negations in simple expressions. It is necessary to also analyze the effect of the number of operators in general, including various combinations of all possible operators. In addition there is much potential in examining different structures of expressions, extending our initial work on adding an external NOT to an expression. Also, numerical expressions and operations on strings may have different characteristics from Boolean expressions, and these too deserve study. Finally, we did not analyze the effect of demographic differences. It would be interesting to see whether experience or linguistic background interact with different levels of understanding.

10 Declarations

10.1 Funding

No funding was received for conducting this study.

10.2 Ethical approval

The School of Computer Science & Engineering Committee for the Use of Human Subjects in Research approved this research on 29 January 2024, for one year. The approval was extended for another year on 8 January 2025.

10.3 Informed consent

We acquired informed consent by presenting the following text in the beginning of the questionnaires: "Participation in the experiment is anonymous and we do not collect any identifying information. The results will be used for research purposes only. You may retire at any point (even though we'd be happy if you stay with us until the end). The experiment is expected to take about 10 to 15 minutes. Continuing to the questionnaire indicates agreement to participate in the research."

10.4 Author Contributions

All authors contributed to the study conception and design. Data collection and analysis were performed by Aviad Baron. The first draft of the manuscript was written by Aviad Baron. All authors reviewed and edited the manuscript, and read and approved the final manuscript.

10.5 Data Availability Statement

All experimental materials and data are available on Zenodo using the DOI 10.5281/zenodo.14970258.

10.6 Conflict of Interest

The authors have no relevant financial or non-financial interests to disclose.

10.7 Clinical Trial Number in the manuscript

Not applicable.

References

1. Galit Agmon, Yonatan Loewenstein, and Yosef Grodzinsky. Negative sentences exhibit a sustained effect in delayed verification tasks. *J. Exp. Psy.: Learning, Memory, & Cognition*, 48(1):122–141, Jan 2022.

2. Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. Syntax, predicates, idioms — what really affects code complexity? *Empirical Software engineering*, 24(1):287–328, Feb 2019.
3. Aviad Baron and Dror G. Feitelson. Why is recursion hard to comprehend? an experiment with experienced programmers in Python. In *Innovation & Tech. in Comput. Sci. Edu.*, volume 1, pages 115–121, Jul 2024.
4. Aviad Baron, Ilai Granot, Ron Yosef, and Dror G. Feitelson. Understanding logical expressions with negations: Its complicated. In *Intl. Conf. Evaluation & Assessment in Softw. Eng*, pages 303–312, Jun 2024.
5. Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. Indentation: Simply a matter of style or support for program comprehension? In *Intl. Conf. Program Comprehension*, number 27, pages 154–164, May 2019.
6. Alan C. Benander, Barbara A. Benander, and Howard Pu. Recursion vs. iteration: An empirical study of comprehension. *J. Syst. & Softw.*, 32(1):73–82, Jan 1996.
7. Ruven E. Brooks. Towards a theory of the comprehension of computer programs. *Int. J. Man Mach. Stud.*, 18(6):543–554, 1983.
8. Lea Budde, Birte Heinemann, and Carsten Schulte. A theory based tool set for analysing reading processes in the context of learning programming. In *Workshop Primary & Secondary Computing Education*, number 12, pages 83–86, Nov 2017.
9. Raymond P. L. Buse and Westley R. Weimer. A metric for software readability. In *Intl. Symp. Softw. Testing & Analysis*, pages 121–130, Jul 2008.
10. Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Trans. Softw. Eng.*, 36(4):546–558, Jul/Aug 2010.
11. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *European Conf. Softw. Maintenance & Reengineering*, number 14, pages 156–165, Mar 2010.
12. Gorka Cesium. Why to stop writing negative code. URL https://medium.com/@Cuadraman/why-to-stop-writting-negavite-code-af5ffb17195, 23 Nov 2018.
13. T. Y. Chen, M. F. Lau, K. Y. Sim, and C. A. Sun. On detecting faults for Boolean expressions. *Softw. Quality J.*, 17(3):245–261, Sep 2009.
14. James M. Clark and Allan Paivio. Dual coding theory and education. *Educational Psychology Review*, 3(3):149–210, Sept 1991.
15. Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. Some empirical resuts for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Australasian Comput. Edu. Conf.*, number 14, pages 77–86, Jan 2012.
16. D. I. De Silva, M. V. N. Godapitiya, Y. S. Kodithuwakku, T. Y. Dewmin, I. H. M. B. L. Dayananda, and K. R. A. W. Fernando. CCMT: A code complexity measuring tool. In *Proc. 9th Intl. Congress Inf. & Commun. Tech.*, pages 101–111. Springer, 2024. Lect. Notes Networks and Systems vol. 1013.
17. Viviane Déprez and M. Teresa Espinal, editors. *The Oxford Handbook of Negation*. Oxford University Press, 2020.
18. I. Deschamps, G. Agmon, Y. Loewenstein, and Y. Grodzinsky. The processing of polar quantifers, and numerosity perception cognition. *Int. J. Man Mach. Stud.*, 143:115–128, 2015.
19. E. W. Dijkstra. Go To statement considered harmful. *Comm. ACM*, 11(3):147–148, Mar 1968.
20. Edsger W. Dijkstra. The humble programmer. *Comm. ACM*, 15(10):859–866, Oct 1972.
21. A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *Intl. J. Human-Computer Studies*, 41(4):457–480, Oct 1994.
22. Dror G. Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software engineering*, 27(6), Nov 2022.
23. Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Trans. Software Eng.*, 48(2):37–52, 2022.
24. Yosef Grodzinsky et al. Logical negation mapped onto the brain. *Brain Structure and Function*, 35:19–31, 2020.

25. Yosef Grodzinsky et al. A linguistic complexity pattern that defies aging: The processing of multiple negations. *Journal of Neurolinguistics*, 58:543–554, 2021.
26. Shuchi Grover and Satabdi Basu. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proc. ACM SIGCSE Technical Symp. Computer Science Education*, pages 267–272, 2017.
27. Bruria Haberman and Haim Averbuch. The case of base cases: Why are they so difficult to recognize? student difficulties with recursion. *ACM SIGCSE Bulletin*, 34(3):84–88, Sep 2002.
28. Geoffrey L. Herman, Michael C. Loui, Lisa Kaczmarczyk, and Craig Zilles. Describing the what and why of students' difficulties in Boolean logic. *ACM Trans. Comput. Edu.*, 12(1), Mar 2012.
29. Laurence R. Horn, editor. *The Expression of Negation*. De Gruyter Mouton, 2010.
30. Nathan Hurtig, Joseph Hollingsworth, Sarah Blankenship, Eileen Kraemer, Murali Sitaraman, and Jason O. Hallstrom. Network visualization and assessment of student reasoning about conditionals. In *Innotaion & Tech. Comput. Sci. Edu.*, number 27, pages 255–261, Jul 2022.
31. Errol R. Iselin. Conditional statements, looping constructs, and program comprehension: An experimental study. *Intl. J. Man-Machine Studies*, 28(1):45–66, Jan 1988.
32. Marcel Adam Just and Patricia Ann Carpenter. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior*, 10:244–253, 1971.
33. Claudius M. Kessler and John R. Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Comput. Interaction*, 2(2):135–166, 1986.
34. Sangeet Khemlani, Isabel Orenes, and P.N. Johnson-Laird. The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica*, 151:1–7, 2014.
35. Daniel Lindner. Don't ever not avoid negative logic. URL https://schneide.blog/2014/08/03/dont-ever-not-avoid-negative-logic/, 3 Aug 2014.
36. Raymond Lister. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Australasian Comput. Edu. Conf.*, number 13, pages 9–18, Jan 2011.
37. Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Innovation & Tech. in Comput. Sci. Edu.*, pages 161–165, Jul 2009.
38. Mike Lopez, Jacqueline Whalley, Phil Robbins, and Rayond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Intl. Comput. Edu. Res.*, pages 101–111, Sep 2008.
39. Thomas J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308–320, Dec 1976.
40. Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proc. 23rd International Conference on Program Comprehension*, number 23, pages 25–35, May 2015.
41. Francisco Moretti. Avoid negative conditionals. URL https://www.franciscomoretti.com/blog/avoid-negative-conditionals, 5 Jun 2023.
42. Isabel Orenes, Linda Moxey, Christoph Scheepers, and Carlos Santamaría. Negation in context: Evidence from the visual world paradigm. *Quarterly Journal of Experimental Psychology*, 69, 2016.
43. Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *Intl. Conf. Softw. Eng.*, number 43, pages 524–536, May 2021.
44. Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Edu.*, 18(1), Oct 2017.
45. Václav Rajlich and George S. Cowan. Towards standard for experiments in program comprehension. In *5th International Workshop on Program Comprehension*, pages 160–161, Mar 1997.
46. Mark Sadoski and Allan Paivio. A dual coding theoretical model of reading. In D. E. Alvermann, N. J. Unrau, and R. B. Ruddell, editors, *Theoretical Models and Processes of Reading*, chapter 34. International Reading Association, 6th edition, 2013.
47. Mark Sadoski and Allan Paivio. *Imagery and Text: A Dual Coding Theory of Reading and Writing*. Routledge, 2nd edition, 2013.

48. Tamarisk Lurlyn Scholtz and Ian Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Innovation & Tech. in Comput. Sci. Edu.*, number 15, pages 103–107, Jun 2010.
49. D. Sleeman, Ralph T. Putnam, Juliet Baxter, and Laiani Kuspa. Pascal and high school students: A study of errors. *J. Edu. Comput. Res.*, 2(1):5–23, Feb 1986.
50. Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Comm. ACM*, 26(11):853–860, Nov 1983.
51. Juha Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Edu.*, 13(2), Jun 2013.
52. Donna Teague and Raymond Lister. Blinded by their plight: Tracing and the preoperational programmer. In *Workshop Psychology of Programming Interest Group*, number 25, Jun 2014.
53. Ye Tian and Richard Breheny. Dynamic pragmatic view of negation processing. *Negation and Polarity: Experimental Perspectives*, 1:21–43, 2015.
54. P. C. Wason. The processing of positive and negative information. *Quarterly Journal of Experimental Psychology*, 11:92–107, 1959.
55. Eliane S. Wiese, Anna N. Rafferty, and Garrett Moseke. Students' misunderstanding of the order of evaluation in conjoined conditions. In *Intl. Conf. Program Comprehension*, number 29, pages 476–484, May 2021.
56. Niklaus Wirth. *Digital Circuit Design for Computer Science Students: An Introductory Textbook*. Springer-Verlag, 1995.
57. Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, 56(4), Apr 2024.
58. Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Softw. Eng.*, 44(10):951–976, Oct 2018.