Not Easy To Understand: Negations in Code and Their Effect on Code Comprehension

Thesis submitted for the degree of "Doctor of Philosophy"

by

Aviad Baron

Submitted to the Senate of the Hebrew University of Jerusalem

8/2025

This work was carried out under the supervision of
Prof. Dror G. Feitelson

Acknowledgment

Abstract

This dissertation investigates the cognitive challenges associated with understanding Boolean expressions in code, focusing on improving code readability and comprehension. The research encompasses three interconnected studies, each exploring distinct aspects of Boolean logic and its effects on program comprehension, while collectively providing a deeper understanding of the factors that influence cognitive processing during code analysis.

The first study examines how specific factors—such as negations, regularity, and truth values—influence the comprehension of logical expressions. In an experiment with 205 professional developers we found that expressions with more negations take more time, and double negations are especially troublesome. However, there are multiple other factors that also have an effect. For example, literals which are TRUE take less time to process than literals that are FALSE. Regularity (where either all variables have negations or all do not, or where either all literals are TRUE or all are FALSE) also helps. But there are many confounding interactions between the factors, leading to complex outcomes.

The second study focuses on the diversity of negations in code. In particular, we considered different types of operators in programming languages that express negation, as well as variable names that contain negation. To explore whether using different negative expressions affects code comprehension, we conducted a controlled experiment involving 268 participants. The task was to understand short code snippets containing various logical expressions and types of negations. Results showed significant differences in comprehension, both in terms of accuracy and time, depending on the type and combination of negations. Complex combinations of different negation types increased cognitive load and reduced correctness. This study emphasizes the importance of simplifying negation patterns in code to improve readability, providing practical insights for refactoring practices and code style guidelines.

The third study addresses a broader distinction in program comprehension by defining and comparing two levels of understanding Boolean expressions: tracing, which involves following the program's execution step-by-step, and comprehension understanding, which requires grasping the overall meaning of the code. In a controlled experiment with 362 participants, the results demonstrated that comprehension understanding took signifi-

cantly longer and was more error-prone compared to tracing. The study further found that expressions involving the logical operator OR were more challenging to comprehend than those with AND, but this difference only emerged at the comprehension level. Additionally, comprehension differences were observed between logically equivalent expressions, with some forms proving more intuitive than others. These findings underscore the complexities of comprehension understanding, which involve deeper cognitive processes and are influenced by subtle syntactic variations. A thorough analysis of the common mistakes shows that they are not random: there is a systematic interaction between the logical OR operator and negation, and when both occur together the logical expression becomes substantially more complex.

Collectively, these studies provide a multi-faceted view of the cognitive challenges involved in Boolean expression comprehension. The findings highlight the significant influence of structural, syntactic, and contextual factors, such as the role of negations, regularity, and operator type, on developers' ability to efficiently and accurately understand code. This work contributes to advancing our knowledge of code readability, offering actionable recommendations for writing clearer, more maintainable code. By examining both practical and theoretical dimensions, it expands our academic understanding of the cognitive processing of negations in the novel and intriguing context of code and suggests opportunities to integrate these findings into modern development practices, including tools for code refactoring and automated readability analysis.

Contents

Abstract **v**

1 Introduction **1**

1.1 Positioning Relative to Related Work 2

1.2 Goals and Research Questions 4

1.3 Structure of the Thesis 6

2 Papers **9**

2.1 Understanding Logical Expressions with Negations: Its Complicated . . . 9

2.2 Is "notDone" the Same as "!done"? The Effect of Different Ways for Expressing Negation 20

2.3 Tracing vs. Comprehension: On Different Levels of Understanding Boolean Expressions 32

2.4 Additional papers 65

3 Discussion and Conclusions **67**

3.1 Contributions 67

3.2 Conclusions 68

References **69**

Chapter 1

Introduction

Negation is a fundamental feature of human language. As linguist Larry Horn writes [24], "In many ways, negation is what makes us human, imbuing us with the capacity to deny, to contradict, to misrepresent, to lie, and to convey irony." Sentence processing in natural language has been investigated with or without negation, with double negations, and with different logical relations, including by using fMRI to map logic processing to brain areas [18, 14, 19, 27, 35, 43, 44, 29, 1]. Such studies shed valuable light on how the human mind processes and comprehends complex logical structures. This is a wide topic: the study of negation was recently surveyed in the *Oxford Handbook of Negation*, which comprises 43 chapters and 756 pages [13].

Program comprehension involves the ability to understand code written by others. It significantly impacts maintenance time as developers construct mental models of the code's structure and functionality [6, 42]. A major barrier to achieving comprehension is the complexity of the code, which plays a pivotal role in determining the ease or difficulty of understanding and maintaining the code. Code complexity in turn is affected by multiple factors, including code length (longer code is more challenging to comprehend), syntactical elements (loops are harder than linear processing), data flow patterns (using indirection makes it hard to analyze), the choice of variable names (to convey clear meanings), and even code layout (e.g. indentation). All of these factors can significantly impact the comprehension of code, either making it more challenging or easier [2, 11, 34, 22].

A fundamental element in any computer program is branching based on logical conditions. The ability to understand and interpret logical expressions is crucial for software development due to their prevalence in code. The insights gained from research on understanding logical expressions and negation in natural language do not necessarily extend to the comprehension of code. Programming languages utilize precise mathematical notation that differs significantly from the nuances of natural language. As a result, issues such as the scope of negation, which are important in linguistic contexts, become irrele-

vant in programming. Moreover, the formal syntax and semantics of code allow for the construction of more complex expressions, such as lengthy formulas involving multiple variables and logical operators, which have no direct counterparts in natural language. In addition, programmers typically possess a strong background in logic and mathematics, making them an unrepresentative sample of the general population.

Surprisingly, research on the comprehension of logical expressions, both in general and specifically pertaining to negation, is nearly absent in the literature on software engineering and code comprehension. There have been recommendations to avoid negations; for example, rule G29 in Bob Martin's *Clean Code* is "Avoid negative conditionals" [31]. But this was not backed by empirical, quantified research. Such research is needed to validate the recommendations, and to address questions about writing more readable code. Also, the issue is more intricate then just avoiding negations. It is important to understand the cognitive mechanisms and cognitive load on developers when comprehending code segments with various logical expressions.

In this work, we examine the difficulty of processing various logical expressions that involve negation. The first part of the thesis focuses on the negation operator itself, investigating how logical negation increases the complexity of understanding logical expressions and identifying structural factors within logical expressions that contribute to this difficulty, including interactions with embedded negations.

In the second study, we delve deeper into negation in the context of programming, exploring the diverse forms of negation in code. This includes different negation operators in popular programming languages, negation in variable names, and the role of negation in conditionals controlling loops.

The third part of the thesis examines the comprehension of logically equivalent expressions across different levels of understanding. We define two levels of code comprehension—"tracing" and "comprehension"—and analyze how logically equivalent expressions are processed at these levels. In the study, we found that certain elements specifically hinder comprehension at particular levels of code processing. Additionally, we discovered the impact of negation on higher-level comprehension, providing insights into why negation presents a significant cognitive challenge.

1.1 Positioning Relative to Related Work

Programming is hard and intellectually challenging [15, 37]. Numerous studies have looked into what makes it hard, especially from the perspective of teaching programming to novices [7, 38, 41, 39]. Many have focused on specific programming constructs, such as loops and recursion [26, 40, 28, 5, 21, 3]. Suggestions for code complexity metrics have been based on counting such constructs [32, 8]. In at least one case, this has included

the logical operators used to create compound logical expressions [12].

However, only few studies have directly addressed the understanding of logical conditions. 37 years ago, Iselin performed an experiment on understanding loops with a condition that either did or did not include a negation, but this was in an operator (equal vs. not equal, in Cobol) and not a logical negation [26]. The only paper we found that studied the understanding of logical conditions with negations is Ajami et al., who compared 3 forms using negation with a similar condition with no negations [2]. The result was that the only highly significant difference in the time to understand the code occurred between 2 negative forms — which were a De Morgan pair — and the 3rd negative form, but no explanation was found for this difference. The conclusion of the study was that "some but not all uses of negation are harder: negations are different from each other".

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 [16]. 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 [23]. 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 [20]. 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, Herman et al. cited above also included an investigation of students' difficulties in writing Boolean expressions [23]. 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 [10]. Several online resources also suggest that developers should avoid negations, and especially double negations, e.g. [9, 33, 30]. 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 various patterns of negations as we do.

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, 4, 36]. Other definitions of understanding were listed by Feitelson [17]. 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 [25]. This thesis is the first comparison between different levels of understanding, specifically in the context of various logical expressions.

1.2 Goals and Research Questions

Our research focuses on understanding the cognitive challenges associated with processing logical expressions, particularly in the context of code comprehension. We investigate various aspects of logical reasoning in programming, including the impact of negations, the complexity introduced by different logical operators, and the gap between comprehension of equivalent expressions. Our work is divided into three main areas, each addressing specific questions related to the difficulty of understanding logical structures in code.

The first aspect of our research examines the influence of the logical negation *operator* on comprehension difficulty. Logical negation plays a fundamental role in Boolean logic, yet it introduces cognitive complexity that can affect code readability and correctness. We explore how the presence of negation operators affects the processing of expressions and whether certain structural factors contribute to this difficulty. Specifically, we investigate:

- **RQ1:** Is there a relationship between the number of logical negations in an expression and the difficulty of its comprehension?
- **RQ2:** How do different logical operators, such as AND and OR, influence comprehension difficulty when combined with negation?
- **RQ3:** Is there a difference in understanding logical expressions based on their truth values (i.e., evaluating to TRUE vs. FALSE)?

The second research direction extends the discussion to different instances of negation in programming languages. While explicit logical negation (e.g., `!`) has been studied in our first research, negation can also appear in variable names (e.g., `isNotEmpty`) or in alternative logical constructs such as inequality operators (e.g., `!=`). We examine whether these different manifestations of negation interact and create additional complexity. To this end, we define the following research questions:

- **RQ4:** What are the different instances of negativity in code?
- **RQ5:** What is the effect of different negation operators, such as `!` versus `!=`, on comprehension?

- **RQ6:** Do negated variable names (e.g., `noErrors`) influence cognitive processing, even though they do not introduce explicit logical negation?

- **RQ7:** How do different forms of negation interact within the same logical condition, particularly in control flow structures such as loops?

Finally, we investigate the comprehension of logically equivalent expressions and how different levels of understanding influence processing difficulty. We define two levels of comprehension: *tracing*, which involves determining the output of an expression for a given input, and *comprehension*, which involves generalizing across multiple inputs. Our findings suggest that expressions that are more readable at one level may be harder to process at another, indicating a gap between tracing and deeper comprehension. In this context, we pose the following research questions:

- **RQ8:** Is there a measurable difference between tracing and comprehension as indicators of code understanding?

- **RQ9:** Are there differences in comprehension between logically equivalent expressions, and do these differences depend on the level of understanding?

- **RQ10:** Do developers utilize their understanding of Boolean expressions to perform "shortcuts" in reasoning, and does this affect comprehension accuracy?

Across all these research directions, we define comprehension in terms of the ability to determine the behavior of code snippets, and we measure difficulty through response times and error rates.

The research contribution of the thesis can be divided into four main parts. The first concerns the cognitive factors affecting how negation and logical conditions are processed. In this thesis, we extended the existing body of research in cognitive science on the comprehension of negation into a new and particularly intriguing context — code. Code offers a unique cognitive landscape, as it combines elements of natural language, such as variable names, with formal mathematical and logical structures, such as operators and conditions. This combination creates a cognitively rich environment, where different types of negation — some rooted in natural language and others in formal mathematical logic — interact and engage the cognitive facilities of the brain. Our findings reveal fascinating cognitive phenomena that appear to be specific to the context of code, including logical and syntactic regularities, as well as the interplay between natural language negation and mathematical-logical negation.

The second dimension of our contribution relates to practical guidelines for writing more readable code, and specifically, for designing clear logical conditions. Our findings identified several factors that increase the complexity of logical expressions, as well as

alternative ways to formulate logically equivalent expressions in a way that enhances their readability.

The third dimension is pedagogical. Writing clear and readable logical expressions is a fundamental skill in software development. Our findings may have important pedagogical implications for how to teach the construction of readable Boolean expressions, offering evidence-based insights into practices that can improve code comprehension from the early stages of programming education.

The fourth dimension is methodological. Our results suggest that code comprehension should be viewed as a layered process, and highlight the importance of distinguishing between different levels of understanding when designing experiments in this domain. It is essential to define the specific level of comprehension being tested, as cognitive factors influencing performance at one level do not necessarily generalize to others.

1.3 Structure of the Thesis

This thesis is structured around three core pillars. The first focuses on the cognitive processing of logical negation as expressed through logical operators. Specifically, we sought to understand how logical negation affects the comprehension of logical expressions. While our results confirmed that the presence of negation tends to increase the cognitive load and complexity of processing, the study also revealed a variety of additional factors that influence the interpretation of logical expressions. Among these are the interaction between negation and the syntactic structure of an expression, including whether the expression exhibits logical regularity — meaning the truth value remains consistent across all literals — and syntactic regularity, referring to whether all variables in the expression are uniformly negated or not. Furthermore, the findings pointed to more intricate interactions that shape comprehension beyond the isolated effect of negation alone.

Importantly, this first line of research focused solely on expressions containing logical negation, without yet addressing the broader variety of negation forms and operators that exist in programming languages. To tackle this, the second pillar of our thesis turns to the world of code, investigating the full range of negation mechanisms: explicit negation operators, logically negated conditions, and the use of negated semantics embedded directly in variable names. In this phase, we systematically explored the diversity of negation in programming, the different ways they are expressed, and the complex interactions between them within real code contexts.

The third pillar addresses the role of comprehension levels in the interpretation of Boolean expressions. While our earlier studies primarily targeted what can be described as the "tracing" level of comprehension — where the focus is on following the logical

structure of an expression — higher levels of understanding also exist. We introduced a framework that distinguishes between different levels of comprehension and specifically investigated how negation impacts the interpretation of Boolean expressions at each level. Our findings highlight intriguing cases where negation significantly increases complexity at certain levels of understanding, and shed light on the underlying cognitive mechanisms. For example, we demonstrated an interaction between the logical OR operator and negation, illustrating how this interaction can explain why the presence of negation sometimes exacerbates the difficulty of understanding other logical operators.

Taken together, this thesis present cognitive factors that underlie the challenges of processing negation in logical expressions, the varied manifestations of negation in the programming context, and the interplay between comprehension depth and the factors which are found to affect the interpretation of Boolean expressions. Our work offers new insights into the cognitive and practical dimensions of code comprehension, with both theoretical and applied implications.

Chapter 2

Papers

2.1 Understanding Logical Expressions with Negations: Its Complicated

Published: Aviad, Baron, Ilai Granot, Ron Yosef, and Dror G. Feitelson. In 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy. ACM.

In “Understanding Logical Expressions with Negations: Its Complicated” we began by examining, at a basic level, the effect of logical negation on the comprehension of Boolean expressions in code. Specifically, we investigated whether there is a relationship between the cognitive difficulty of processing an expression and the number of negations it contains. We also examined the impact of the logical operators used in expressions, and their truth values.

This study opened the door to identifying a range of cognitive factors that influence how logical conditions are processed. Our findings reveal that the picture is far more complex than initially assumed, with significant interactions among multiple factors. The central insight of this paper lies in identifying two types of ”regularities” which affect the difficulty of processing code: logical structural regularity, referring to a recurring pattern in the syntactic form of literals, and logical truth-value regularity, referring to a recurring pattern in the truth values of literals within an expression.

Understanding Logical Expressions with Negations: Its Complicated

Aviad Baron
aviad.baron@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Ilai Granot
Ilai.Granot@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Ron Yosef
Ron.Yosef@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University
Jerusalem, Israel

ABSTRACT

The flow of control in computer programs is shaped by conditional branches. The Boolean expressions which determine the outcome of a branch may have an effect on the readability of the code. In particular, negations can make such expressions harder to understand. We conduct an experiment with 205 professional developers who needed to understand different logical expressions. The results show that the time needed to understand different expressions of similar size can vary significantly. In general, expressions with more negations take more time, and double negations are especially troublesome. However, there are multiple other factors that also have an effect. For example, literals which are TRUE take less time to process than literals that are FALSE. Regularity (where either all variables have negations or all do not, or where either all literals are TRUE or all are FALSE) also helps. But there are many confounding interactions between the factors, leading to complex outcomes. For example, when comparing De Morgan's logically-equivalent pairs of expressions, we found that understanding a negated OR took slightly more time than the AND of two negations, but there was no difference between a negated AND and the OR of two negations. The factors we identified as influencing the understanding of expressions may contribute to advancing our knowledge of cognitive processes involved in understanding logical expressions, but much additional work is still needed. At the same time, the comparisons of equivalent forms provide some practical advice on how to write more understandable expressions.

CCS CONCEPTS

• **General and reference** → **Design**; *Experimentation*; • **Theory of computation** → *Programming logic.*

KEYWORDS

Code comprehension, Logical expression, Negation

EASE 2024, June 18–21, 2024, Salerno, Italy

ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661180

ACM Reference Format:
Aviad Baron, Ilai Granot, Ron Yosef, and Dror G. Feitelson. 2024. Understanding Logical Expressions with Negations: Its Complicated. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661180

1 INTRODUCTION

Program comprehension involves the ability to understand code written by others. It significantly impacts maintenance time as developers construct mental models of the code's structure and functionality [3, 18]. A major barrier to achieving comprehension is the complexity of the code, which plays a pivotal role in determining the ease or difficulty of understanding and maintaining the code. Code complexity in turn is affected by multiple factors, including code length (longer code is more challenging to comprehend), syntactical elements (loops are harder than linear processing), data flow patterns (using indirection makes it hard to analyze), the choice of variable names (to convey clear meanings), and even code layout (e.g. indentation). All of these factors can significantly impact the comprehension of code, either making it more challenging or easier [2, 4, 10, 15].

A fundamental element in any computer program is branching based on logical conditions. The ability to understand and interpret logical expressions is crucial for software development due to their prevalence in code. Logical expressions in general and negation in particular have long captivated the minds of scholars across various disciplines, including logicians, philosophers, linguists, and psychologists. Sentence processing in natural language has been investigated with or without negation, with double negations, and with different logical relations, including by using fMRI to map logic processing to brain areas [1, 6, 8, 9, 12, 13, 16, 19, 20]. Such studies shed valuable light on how the human mind processes and comprehends complex logical structures. This is a wide topic: the study of negation was recently surveyed in the *Oxford Handbook of Negation*, which comprises 43 chapters and 756 pages [5].

But prior research on understanding logical expressions and negation in natural language does not necessarily carry over to code comprehension. For one thing, programs use unambiguous mathematical notation which is different from the facilities of natural language; thus discussions on issues like the scope of negations are irrelevant. The formal syntax and semantics also allow more

complex expressions to be used: in code one can find long formulas involving multiple variables and logical operators, but there are no such constructs in natural language. In addition, programmers are typically well-versed in logic and mathematics, and are therefore not representative of humans in general. But investigating the comprehension of code-related logical expressions may shed light on how the human brain processes negation and complex logical constructs in a unique context.

Surprisingly, research on the comprehension of logical expressions, both in general and specifically pertaining to negation, is nearly absent in the literature on software engineering and code comprehension. There have been recommendations to avoid negations; for example, rule G29 in Bob Martin's *Clean Code* is "Avoid negative conditionals" [14]. But this was not backed by empirical, quantified research. Such research is needed to validate the recommendations, and to address questions about writing more readable code. Also, the issue is more intricate then just avoiding negations. It is important to understand the cognitive mechanisms and cognitive load on developers when comprehending code segments with various logical expressions.

36 years ago, Iselin performed an experiment on understanding loops with a condition that either did or did not include a negation, but this was in an operator (equal vs. not equal, in Cobol) and not a logical negation [11]. The only paper we found that studied the understanding of logical conditions with negations is Ajami et al., who compared 3 forms using negation with a similar condition with no negations [2]. The result was that the only highly significant difference in the time to understand the code occurred between 2 negative forms — which were a De Morgan pair — and the 3rd negative form, but no explanation was found for this difference. The conclusion of the study was that "some but not all uses of negation are harder: negations are different from each other".

The long-term goal of our work is to uncover the cognitive processes of processing and reasoning involved in comprehending logical expressions and negations among professional programmers. As a beginning we start by looking for logical and structural factors that might impact their efficacy in this domain. In this we follow the work of Ajami et al., who investigated various factors of code complexity beyond negations [2]. Specifically, in our work we tried to dissect logical expressions into basic components, and investigate the influence of various combinations on code comprehension, with a specific focus on negations.

Our methodology is based on a controlled experiment, in which participants were asked to find the output of different code snippets that were crafted specifically for the experiment. The participants were 205 professional developers from around the world. The code snippets contained various logical expressions, and we assessed their understanding and ability to correctly interpret the code.

The experiment consisted of two main parts. In the first part, we examined how the number of negations in a logical expression affects the difficulty of understanding the expression. We also considered the interaction between this phenomenon and the type of logical operators (AND or OR) and the expression's truth value. Our findings demonstrate that the number of negations in the code snippet indeed has an impact on code comprehension. However, there is also an influence of the structural arrangement of the code and the recurring patterns within it, and intriguing complex interactions between them.

The second part of the experiment aimed to examine equivalent logical expressions and compare their levels of comprehension difficulty. The expressions we examined included, among others, equivalent ways of stating a logical expression using De Morgan's laws, using double negations, and more. The results also revealed complex interactions among various factors that influence the processing difficulty in this context.

Our contributions in this paper are:

- The first in depth investigation of comprehending logical expressions in program code, and specifically the effect of negations. This extends existing research on negation in natural language to a completely new context.
- The identification of structural and logical factors, and their effects on understanding logical expressions in code. These factors include
  - The number of negations in an expression;
  - The truth value of each literal[1];
  - The regularity of the expression in terms of syntax (all variables have negations or none do) or logic (all literals are TRUE or all are FALSE).
- Showing the existence of multiple complex interactions between these factors. This implies that it may be hard to describe the full cognitive processes underlying the understanding of logical expressions with negations.

2 RESEARCH QUESTIONS

Our work concerns the comprehension of different kinds of logical expressions, and specifically, comparing equivalent logical expressions, logical expressions with negations, expressions with different logical operators, and expressions with different truth values. Within this context, our concrete research questions are:

(RQ1) Is there a relationship between the number of logical negations in an expression and the difficulty of the processing?

(RQ2) Is there any difference in comprehension between an expression that combines literals with the logical operator AND and an expression that combines literals with the logical operator OR?

(RQ3) Is there any difference in comprehending a logical expression with a truth value of TRUE compared to comprehending a logical expression with a truth value of FALSE?

(RQ4) Are there other factors which influence the processing of logical expressions in code?

(RQ5) Is there any difference in understanding between two logically equivalent expressions? Specifically, is there a difference in understanding De Morgan's equivalent pairs — for example, the negation of a conjunction of variables compared to the disjunction of the negated variables?

In all these questions, comprehension is defined as finding what a code snippet prints, and the metrics for difficulty are the time this took and the fraction of wrong answers. We leave the question of

[1]To clarify our terminology: 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.

how all this depends on different definitions and metrics for future work.

3 EXPERIMENTAL DESIGN AND EXECUTION

The experiment included two sections. In both, participants were presented with multiple code snippets for comprehension. Each snippet commenced with the declaration and initialization of several Boolean variables. Subsequently, these variables were employed in an expression within an `if` statement, leading to the printing of one of two distinct strings. The objective was to ascertain which string would be printed. The code snippets were written in Python, which is one of the most popular programming languages today. It also has the advantage that logical expressions are very transparent and readable, for example using `not` for negations rather than ! as in C.

3.1 Code Snippet Considerations

We took several methodological considerations into account in designing the experimental materials and the execution of the experiment, with the goal of reducing threats to validity [7].

A basic decision was to avoid situations where the evaluation could be influenced by intuition. For example, we initially thought of using code that has an everyday appeal, such as the following:

```
is_summer = True
eating_ice_cream = False
if is_summer and eating_ice_cream:
    print("happy")
else:
    print("sad")
```

But we decided not to use such codes, because they have the drawback that participants might be influenced by the semantics of the described situation. For example, most people would probably associate the condition `is_summer and eating_ice_cream` with the output string `"happy"`. Given this condition they might then conclude that this is the correct answer irrespective of the actual logic of the code. And if we wanted to use a negation and had written `is_summer and not eating_ice_cream` this might create a cognitive dissonance with the following instruction `print("happy")` thereby making the code more difficult. To avoid this we used code that talks about colors, geometric shapes etc. — namely variables that are independent of each other, and do not have any marked intuitive associations like ice cream with summer. This applies both to the variables and to the outputs of the code snippets.

Another possible problem with the above example is that the lengths of the outputs are different. This may cause a bias in the results, because what the participants are asked to do is to write the expected output. To avoid this we used single-letter outputs, A and B.

A third methodological consideration was to require that participants had to read the entire condition in order to infer its truth value. Our first research question RQ1 concerns the effect of the number of negations in an expression. We therefore need to avoid the danger of "short circuits". For example, if the first literal out of three literals connected by ORs is TRUE, you immediately know that the whole expression is TRUE, regardless of the values of the other two literals — and regardless of whether or not they are negated. If this happens, we won't know whether a shorter response time was due to the expression having fewer negations, or to the expression not being read to the end. In other words, the option of short-circuiting is a major confounding variable. We therefore selected the truth values of the literals such that all three must be considered to reach a conclusion. This approach ensured that the comparison was indeed between complete conditions with three literals, without the confounding effect of conditions that could be shortened.

We note that this last design choice may create a different threat. Having to read the whole expression implies that the last literal is actually the one that determines the result. If participants notice that the evaluation always hinges on the last literal, they may be tempted to skip directly to this literal and ignore the previous ones. We believe that the danger that this happens is low, for two reasons. First, it is hard to notice such structure in a relatively short experiment where we gave each participant only part of the complete set of code snippets. Second, even if participants suspect this feature of the design, they would probably still check that it indeed always holds. Furthermore, even in the event that some participants decide to skip the initial literals, the randomization of question order implies that there will be no systematic effect. Therefore we believe this design is worth the price to ensure that short circuits are not taken and all negations are read.

3.2 Experiment Execution

The experiment was conducted using the Qualtrics surveys platform. In executing the experiment we randomized the order of the code snippets presented to the participants, to mitigate the effects of fatigue, cognitive bias, and other confounding factors. By employing randomization, we ensured that on average participants independently and individually saw different code snippets before or after other snippets, without systematic biases.

The experiment started with an introductory page explaining what the experiment is about. This included details about the number of trials, the approximate time the experiment is expected to take, and a general overview of the experiment's purpose: "Our goal is to understand the cognitive mechanisms of reading different logic patterns in code". Informed consent to participate was explicitly reflected by moving to the next page.

The experiment itself consisted of two parts with a total of 15 code segments. The first part included 16 code segments, constituting a full factorial design to compare the effect of different levels of different factors (the number of negations, using AND as opposed to OR, etc., as detailed below). Each participant received 8 of these 16 segments, chosen randomly, and in random order. The randomized choice implies that the comparisons are between subjects. Part two included another 7 code segments, designed to compare pairs of equivalent expressions (3 pairs and one control, detailed below). In this case each participant received all 7 in random order. Because each participant saw all the codes the comparison in this part is within subjects. The response time for each question is measured automatically by Qualtrics. This is the time from when the page with the code snippet is presented until the participant clicks on the "next" button after entering the answer.

The participants were recruited through various channels, including WhatsApp groups of programmers, online forums on reddit

(which proved most effective), and direct recruitment efforts. A total of 205 participants took part in the experiment. No identifying information was collected, but we did ask basic demographic questions. 163 of the participants reported their gender: 160 of them were men and 3 were women. This is a rather extreme ratio, but quite similar to that observed in the Stack Overflow developer survey[2]. It is therefore actually representative of the developer community. 168 participants reported their academic background. Among them, 35 had no formal academic background, 92 had a BSc degree, 35 had an MSc degree, and 6 had a PhD. Out of the 162 participants who reported their years of experience, 19 had 0–2 years of experience, 74 had 3–10 years of experience, and 69 had over 10 years of experience. These numbers indicate that the participants are mostly rather experienced developers. Those with little experience are few and are not expected to have an appreciable effect on the results. We did not collect data about the participants' domain of work or programming languages, as we are dealing with very basic constructs which are present in similar form in all imperative programming languages.

4 THE EFFECT OF NEGATIONS AND CONTEXT

The first part of the experiment was designed to answer Research Questions RQ1 through RQ3. To achieve this goal, participants were presented with various code snippets, which included different combinations of levels of three factors:

- Having different numbers of negations in the logical expressions. This included the base case of no negations, which serves for comparison. In other words, we collect data both about *having* negations and about *the number* of negations.
- Having different logical operators — either AND or OR.
- Cases where the entire logical condition evaluates to TRUE versus cases where it evaluates to FALSE. This was achieved by correctly setting the truth values of the variables in the conditions.

By comparing participants' responses and performance across these diverse code snippets, we were able to assess the impact of the different factors on cognitive load and investigate the relationship between them and code comprehension difficulty. We also studied the results to see if we could identify any additional factors that may have an effect, to answer Research Question RQ4.

4.1 Experimental Materials

Each code snippet in this part of the experiment contained a logical expression with exactly three variables, so that the length of the expression would not be a factor. The variables were initialized before the logical expression, in the same order that they appeared in the expression. Each variable could potentially have a logical negation or not. As each expression contained three literals, the number of negations in the condition ranged from 0 to 3. This allows us to see whether more negations in the expression lead to higher cognitive load and reduce performance.

We decided that all the negations will be placed consecutively on the last variables: if there was one negation is was on the last variable, and if two they were on the last two variables. This was done to eliminate a potential confounding factor — the effect of mixing variables with and without negations.

The logical connectives between the literals could be either both AND or both OR. In other words, we deliberately maintained distinct contexts for the logical operators, and do not investigate the effect of mixing them. Additionally, a potentially important factor we examined was whether the entire logical condition evaluated to TRUE or FALSE.

Putting all of this together leads to 16 combinations: 4 options for the number of negations (0 to 3), multiplied by two options of the logical operator used, multiplied by two options for the final result. For example, the following code snippet represents the combination of 1 negation, the logical operator OR, and a FALSE expression value:

```
is_green = False
is_card = False
is_circle = True
if is_green or is_card or not(is_circle):
    print("A")
else:
    print("B")
```

As another example, in the following code segment there are 3 negations, the logical operator is AND, and the truth value is TRUE:

```
is_blue = False
is_wet = False
is_card = False
if not(is_blue) and not(is_wet) and not(is_card):
    print("A")
else:
    print("B")
```

4.2 Results

For each code snippet we have two results: the fraction of participants who understood it correctly, and the time it took them to do so. In the figures we show the CDF (cumulative distribution function) of the time. The time is on the horizontal axis, and the graph shows the probability to solve the problem 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. The scale is truncated at 40 seconds, because most of the code snippets are very simple and take only 10–20 seconds to interpret. This excludes the few outliers that may be present.

The graphs include the correctness results by giving wrong answers an infinite time. As a result the CDFs do not reach a maximal value of 1, but rather the fraction of correct answers. But because the code snippets are rather easy, the participants nearly always answered correctly, so this was usually 1 or very close to 1. The number of wrong answers was too small to allow for meaningful analysis. In the sequel we therefore focus on differences in the time needed to answer, and not on correctness.

Figures 1 and 2 show all the results, for expressions with the AND operator and the OR operator respectively. These are rather dense, but still some interesting effects can be seen. For example, in Figure 1 we can see that expression using AND that has no

[2]https://survey.stackoverflow.co/2022#developer-profile-demographics

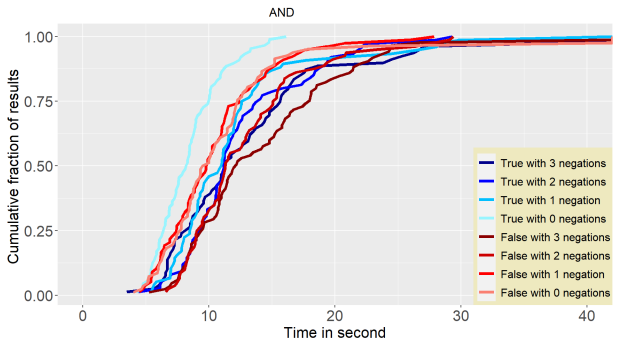


Figure 1: CDFs of the time to correct answers for logical expressions with 3 literals connected by AND operators.

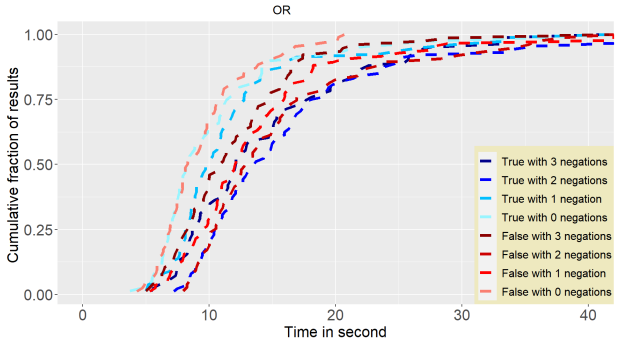


Figure 2: CDFs of the time to correct answers for logical expressions with 3 literals connected by OR operators.

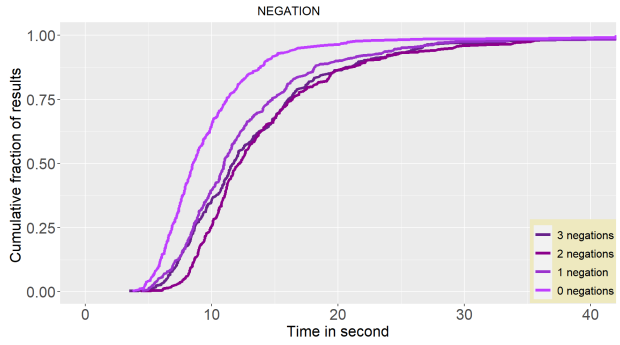


Figure 3: CDFs of the time to correct answers for logical expressions with different numbers of negated variables.

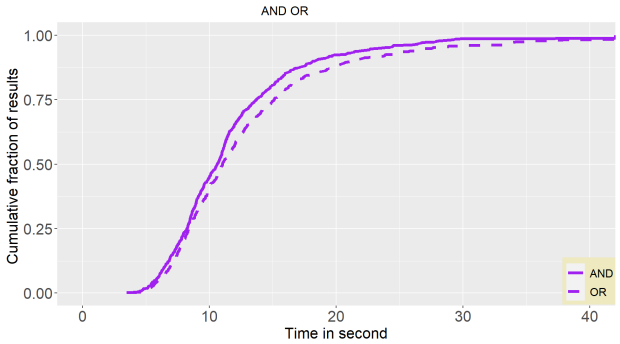


Figure 4: CDFs of the time to correct answers for logical expressions with different operators.

negations and evaluated to TRUE was the fastest to interpret, but the same expression that evaluated to FALSE was no faster than an expression with a negation. But in Figure 2 we see that such an effect does not exist for expressions with OR. On the contrary, there is a bigger difference when the expressions evaluate to FALSE compared with when they evaluate to TRUE.

Such observations imply that there are various complex interactions between the factors. To uncover them we start by looking at the effect of each factor separately. We then move to discussing the interactions in Section 4.3.2.

4.2.1 Effect of Number of Negations. We start with the factor of the number of negations, which is the subject of Research Question RQ1. Figure 3 shows the results for the processing time as a function of the number of negations in the logical expression, for all operators and truth values (that is, each line in this graph contains data from 4 lines in Figures 1 and 2). The results demonstrate that having more negations increases the response time. When comparing a logical expression without any negations to a logical expression with a single negation, and when comparing a logical expression with a single negation to a logical expression with two negations, the addition of the negation increases the time. However, this is not the case when comparing an expression with two negations to an expression with three negations. This effect is the result of an interaction that will be discussed later.

More formally, the independent variable has four levels, representing the number of negations. The dependent variable is the time of responses. The comparison is between subjects. We would like to check whether the average time needed to process the different versions (in pairs) is equal. For this we will use 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 these tests is that there is a statistically significant difference between no negations and 1 negation (p-value = 3.173e-13), and also between 1 negation and 2 negations (p-value = 0.0001126). So in these two cases the null hypothesis is rejected. However, the difference between 2 negations and 3 negations is not statistically significant (p-value = 0.07602). Thus the null hypothesis was *not* rejected in this case.

4.2.2 Effect of AND vs. OR and TRUE vs. FALSE. The additional factors we considered, in order to examine Research Questions RQ2 and RQ3, are the logical operator and the value of the logical condition.

In order to examine RQ2 we draw Figure 4 which shows the results of processing times for conditions that use the AND operator and conditions that use the OR operator. These are combined results for expressions with all the different numbers of negations (0 to 3) and different truth values. As can be seen in the graph a small difference is observed: conditions using AND take slightly less

Figure 5: CDFs of the time to correct answers for logical expressions with different truth values.

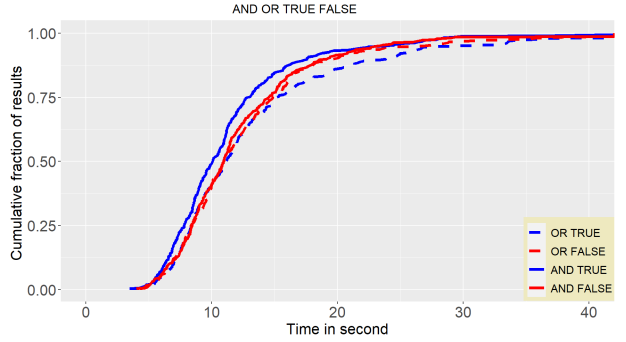


Figure 6: CDFs of the time to correct answers for logical expressions with different operators and truth values.

time. Applying the t-test as previously showed that the difference is statistically significant, with p-value = 0.001048. But it seems that the effect of the operator factor is not very meaningful, as the actual difference in times is slim.

Figure 5 shows the results of processing times for conditions that evaluate to TRUE and conditions that evaluate to FALSE, combining the results for expressions with different numbers of negations and different operators. It is evident that they are practically the same. Therefore the condition's truth value is not an important factor in and of itself. This observation is also supported by the statistical test, which had a p-value = 0.7401.

However, the actual picture is more complicated. Figure 6 shows the results for the four combinations of operator and truth value. What we can see is that when the condition is FALSE (red lines), there is no difference between conditions using AND and conditions using OR (p-value = 0.3293). The difference we saw above in Figure 4 is wholly due to conditions that evaluate to TRUE (p-value = 0.0003977).

In other words, the effect we saw was not an effect of the operator factor, but an effect of the interaction between the operator and the truth value.

In summary, of the three independent variables we started with — the number of negations, the operator, and the truth value — only the number of negations seems to have an effect on the processing

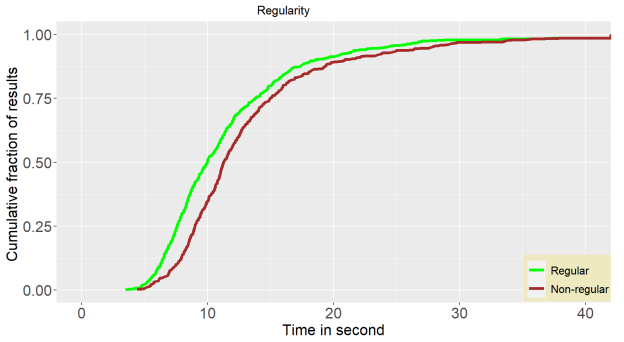


Figure 7: CDFs of the time to correct answers for regular logical expressions vs. irregular logical expressions.

time by itself. The factors of operator and truth value do not. However, an *interaction* between these two factors does have an effect. In the following section, we propose additional factors that may be the real causes of this interaction.

4.3 Identification of Additional Factors

In designing the experiment we made two methodological decisions which may have had repercussions we did not anticipate in advance. These decisions were the following:

- We wanted the participants to always read the whole condition, without the option of short circuiting. For a condition based on ANDs this implies that all the initial literals be TRUE. For a condition based on ORs this implies that all the initial literals be FALSE. In both cases, the last literal then determines the truth value of the whole condition.
- We decided to group all the negations together, and do this consistently in all the different conditions. Specifically we decided to always put them at the end of the condition (but we could also have decided to put them in the beginning). In this way the distribution of negations will not be an additional confounding factor.

Our analysis indicates that these decisions inadvertently created additional factors that influence the results, thereby answering Research Question RQ4.

4.3.1 The Factors. We start by describing the additional potential factors we found. One additional factor we propose came about as a result of the above design decisions is *syntactic regularity*. Syntactic regularity refers to the literals in the condition having the same structure: either none of them are negated variable or all of them are negated variables. The alternative (irregularity) is to have a mix, where only part of the variables are negated. Our interpretation is that this factor explains the anomaly seen above in Figure 3, where conditions with 3 negations were shown to take similar or less time that conditions with 2. Specifically, conditions with 3 negations are regular, and we believe this compensates for the difficulty caused by the additional negation.

To verify this, we drew a graph that compares all the regular conditions with all the irregular ones. The regular conditions are those with 0 or 3 negations, and the irregular ones are those with

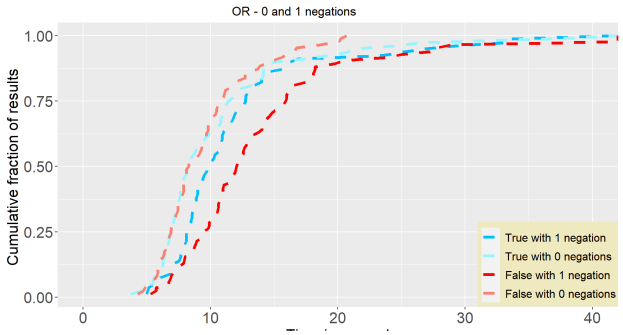


Figure 8: CDFs of the time to correct answers for expressions based on OR with 0 or 1 negations.

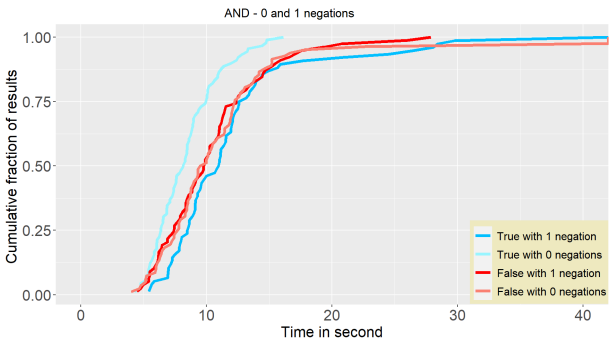


Figure 9: CDFs of the time to correct answers for expressions based on AND with 0 or 1 negations.

1 or 2 negations; in either case, the average is 1.5 negations. As we can see in Figure 7, the regular ones indeed take less time to understand. Formally the independent variable is categorical with two levels, whether the expression is regular or not. The dependent variable is the time of responses. The null hypothesis in the t-test is that the times are equal, and the alternative hypothesis is that the expected values are different. The results are that there is a statistically significant difference, with p-value = 6.205e-07. Thus the null hypothesis was rejected.

The regularity defined above is syntactic: it concerns the structure of the condition. Another form of regularity is *logical regularity*. This refers to whether or not the literals have the same truth value, namely whether they are all TRUE or all FALSE.

The evidence supporting this factor comes from Figure 6. In that figure we saw that conditions using AND which evaluate to TRUE took less time to understand than similar conditions that evaluate to FALSE. Our interpretation is that those that evaluate to TRUE are necessarily composed of 3 TRUE literals in a row, whereas in those that evaluate to FALSE the last literal is different.

Note that this effect is not observed in conditions based on OR: in this case conditions that evaluate to TRUE and conditions that evaluate to FALSE take about the same time. To explain this we introduce a third possible factor, which is the truth value of the individual literals: whether each one of them is TRUE or FALSE. When conditions use AND, the two last factors work together: conditions that are logically regular also have more TRUE literals. But for conditions based on OR the two factors counteract each other: those that are regular have *fewer* TRUE literals. We suggest that this is why we do not observe a difference in the distributions of the total time to understand these conditions. As further support for this third factor, note that in the figure the expression with AND that evaluate to TRUE took slightly less time than those with OR that evaluate to FALSE, despite both having the same level of logical regularity. We explain this by the fact that for AND all the literals evaluate to TRUE, while for OR they all evaluate to FALSE.

4.3.2 Interactions. The above factors do not tell the whole picture. In addition there are interactions between them. We illustrate this by comparing just the conditions with no negations and a single negation. These are shown in Figures 8 and 9. The factors and their levels are summarized in Table 1.

			regularity			
op	neg	truth	syn	log	T-lit	result
OR	0	TRUE	✓		1	no gap
	0	FALSE	✓	✓	0	
AND	0	TRUE	✓	✓	3	gap
	0	FALSE	✓		2	
OR	1	TRUE			1	gap
	1	FALSE		✓	0	
AND	1	TRUE		✓	3	no gap
	1	FALSE			2	

Table 1: Summary of interacting factors. (T-lit = number of literals that are TRUE)

Let us first summarize the main results. When conditions use the OR operator, and there are no negations, we find that there is no difference between TRUE and FALSE conditions. But when the conditions contain one negation there is a significant difference in favor of the TRUE condition. With the AND operator this result is reversed: the difference between TRUE and FALSEoccurs for conditions with no negations, but not in conditions with a negation.

We explain these differences as follows. In OR with no negations the TRUE version has one more TRUE literal, while the FALSE version has logical regularity. These effects cancel out and the results converge with no gap. In AND with no negations the TRUE version has both one more TRUE literal *and* logical regularity. This double advantage over the FALSE version causes a gap to appear.

When conditions have a negation the trend is changed for both OR and AND. Recall that we placed the negation on the last variable. As a result in the combinations of TRUE OR and FALSE AND the last literal is changed in two ways at once: it gets a negation, and it also changes its truth value. We believe that his simultaneous change of the syntax and the semantics in the same literal aids comprehension. As a result a gap is formed for OR and the gap is closed for AND. With 2 or 3 negations the picture changes again, and is more similar to the situation with no negations.

This demonstrated the complexity of the situation: syntactic regularity usually aids comprehension, but breaking it in tandem with a break in logical regularity can be beneficial too.

The above analysis attempted a first mapping of the possibly relevant factors and their effects. However a full picture will only be possible after conducting multiple additional experiments, which will be specifically designed to study the ideas we raised here. For example, our definition of regularity is dichotomous: we require either all or none of the literals to have negations. It is interesting to also check what happens in between, with different fractions of literals with negations.

5 COMPARING EQUIVALENT FORMS

In the second part of the experiment, our objective was to compare equivalent logical expressions to examine which version is more readable and comprehensible, in order to answer Research Question RQ5. For example, this included comparing equivalent expressions related by De Morgan's laws. We aimed to determine which form of the logical expressions is more easily understood by the participants.

5.1 Experimental Materials

The code snippets used in this part again include Boolean variables initialized to their respective Boolean values, and then a logical condition that uses them. For example, the following code represents the logical condition $\neg(p \wedge q)$:

```
is_pink = True
is_circle = False
if not(is_pink and is_circle):
    print("A")
else:
    print("B")
```

The participants are then asked to respond with what will be printed, based on the given code and variable assignments.

The snippets were designed to capture various pairs of equivalent formulations. For example, the above snippet has an alternative version using a logical expression equivalent to the previous one according to De Morgan's laws, namely $\neg p \vee \neg q$:

```
is_pink = True
is_circle = False
if not(is_pink) or not(is_circle):
    print("A")
else:
    print("B")
```

Furthermore, we examined the option of using an *implicit* negation, meaning the condition was not satisfied, and thus the code executed the "else" statement rather than the "then" statement. This is equivalent to using an explicit negation and switching the "then" and the "else". For example, the following expresses exactly the same logic as the first snippet shown above, without using any negation:

```
is_pink = True
is_circle = False
if is_pink and is_circle:
    print("B")
else:
    print("A")
```

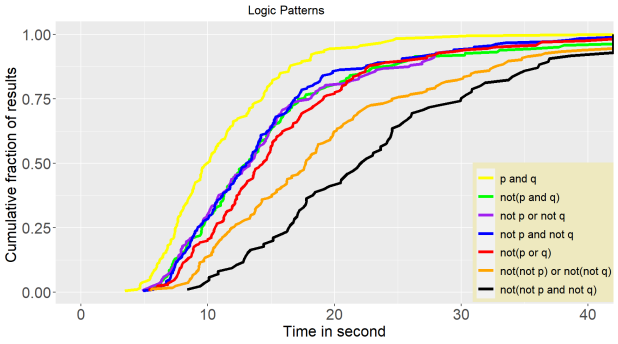


Figure 10: CDFs of the time for comprehending the seven snippets. All had the value TRUE except $p \wedge q$, which was FALSE and represents an implicit negation.

(However, in order not to give this away, in the experiment we did not switch the printing of A and B.)

All told we used seven code snippets which implement the following logical conditions:

- 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 $\neg p \wedge \neg q$
- The "implicit negation" when the condition is simply $p \wedge q$ but it receives a false value.
- Expressions with double negations, represented by the condition $\neg(\neg p \wedge \neg q)$ and its corresponding expression $\neg\neg p \vee \neg\neg q$.

In the analysis of the results we compared various pairs of expressions to see the effect of their different structures.

The methodological considerations noted in the previous part of the experiment were also used in this part. In this part, all participants saw all seven conditions in a randomized order.

5.2 Results

Figure 10 shows the time distributions measured for the seven snippets, with wrong answers represented as ∞ as before. Obviously there are significant differences between them.

In the results, we observe that explicit negations increase cognitive load (as reflected by the time needed to produce an answer). The fastest condition to understand was the one with the "implicit negation", namely $p \wedge q$ that evaluated to FALSE. This was significantly faster than the logically equivalent $\neg(p \wedge q)$ that evaluated to TRUE, but had a negation (t-test p-value = 7.803e-07).

In general, all four expressions: $\neg(p \wedge q)$, $\neg p \vee \neg q$, $\neg(p \vee q)$, and $\neg p \wedge \neg q$ are quite similar in terms of their processing difficulty, with a slightly higher difficulty for the condition $\neg(p \vee q)$. Performing the t-test on De Morgan pairs, we find that there are no statistically significant differences. For the pair $\neg(p \wedge q)$ and $\neg p \vee \neg q$, which look like exactly the same time distribution, the result was p-value = 0.5916. But also for the pair $\neg(p \vee q)$ and $\neg p \wedge \neg q$, where the graphs show some difference, the result was *not* significant, with p-value = 0.0715. Note that this is a within-subject analysis.

The implication is that more negations is not always worse — it also depends on what exactly is negated: is it a variable or a

more complex expression. In particular, it seems that the equivalent forms of De Morgan's laws are similar, because while one has two negations and the other only one, that one negation is applied to a compound expression, and the effects cancel.

A similar effect is seen in the conditions that include double negations. First, we note that both these conditions were the hardest to understand. They took the longest time to understand, and were also the only code snippets in our experiments where the number of mistakes was not negligible. Note that the condition $\neg\neg p \vee \neg\neg q$, which has 4 negations, took less time than the equivalent expression $\neg(\neg p \wedge \neg q)$ that has only 3 (p-value = 0.002088). We believe this is because of the confluence of three effects: the second condition has a negation that applies to a complex expression; the first expression has structural regularity, with $\neg\neg x$ appearing twice; and it is easy to see that the double negations in the construct $\neg\neg x$ cancel out.

To summarize, we find that in some cases there may be significant difference in processing of conditions that are actually logically equivalent, but in other cases there is no large difference. In other words, logical equivalence does not inherently guarantee that the logical conditions will have the same processing difficulty. What holds more significance is the logical and structural composition of each logical condition.

6 THREATS TO VALIDITY

Construct validity. We wanted to measure the difficulty in understanding logical expressions. However, there are many different levels of understanding [7]. We measured the ability to follow and understand what the expression prints, which reflects an understanding of the programming language and an ability to trace the execution of the code. It does not necessarily reflect a higher level of understanding. However, when using short code snippets in an attempt to isolate specific factors, as we do here, there is no real options to create "meaningful" code that justifies such higher levels. We therefore contend that this choice is appropriate.

Concerning the difficulty in understanding, 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 [17]. However, while they 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. Our interpretations of the results are at times somewhat speculative, as additional potential factors were identified during the analysis that were not anticipated in advance. Additional experiments need to be designed and executed to further validate these factors and their effects.

We designed the experiment such that the participants need to read the full expressions, and therefore the last literal is the decisive one. It is possible that participants may have discerned the significance of the last literal in the code, and skipped other parts of the expression. However, given the brevity of the experiment, we believe this risk is minimal, especially since we shortened the experiment and gave each participant only 8 of the 16 snippets in the first part of the experiment. It is also unlikely that participants in an experiment would be so sure of themselves that only the last literal matters that they will skip the initial ones. In addition, because we randomized the presentation of the code snippets, any effect (if it exists) would be spread evenly across all the codes and there will be no systematic effect. We therefore believe this threat is not significant.

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 basic formulas. It is important to acknowledge that the results for the expression we employed may not necessarily generalize to other scenarios or expressions. There is no alternative to performing additional experiments to get a fuller picture.

One specific example concerns the use of logic short-circuiting. We designed the logical expressions such that short-circuiting is not possible. This was required in order to ensure that we are comparing the reading of expression of the same length. At the same time, we acknowledge that the practice of short-circuiting logical expressions does exist. We are planning to conduct experiments about the use and effects of short-circuiting in the future.

7 CONCLUSION

Understanding logical conditions in code is complicated. We believe this research has demonstrated this complexity. It suggests that many factors are involved in this activity.

We showed that the time needed to understand logical expressions is affected by numerous factors and interactions among them, influencing the processing difficulty. We sought to characterize and identify some of these factors, which can be broadly categorized into two main dimensions: *syntactic factors* and *logical factors*. For example, the negation operator is part of the syntactic dimension, while a literal's truth value is in the logical dimension. And both of these factors can impose processing difficulties. Additionally, both syntactic and logical *regularities* are also factors influencing processing difficulty. Furthermore, we discovered that the factors are not independent, and there are interactions between the syntactic and logical factors. For example, when syntactic regularity and logical regularity brake down simultaneously in the same literal, the correlation between the syntactic and logical perspective eases the processing, despite the absence of both regularities.

The factors we identified in this study can aid in understanding what influences the comprehension of logical conditions in code, both in a general sense and, more specifically, how they impact the writing and comprehension processes of developers. Beyond that, we believe that this research may contribute to a broader understanding of the cognitive processes associated with the comprehension of complex logical statements. There has been a lot of previous work on understanding negations and logic expressed in natural language. Our results are different in that the expressions are not expressed in natural language, but in a formal notation of programming. This may eliminate some of the ambiguity present in natural language, and enable a sharper focus on the core effects of the logical constructs. In addition, code may expose new factors

that are not commonly observed in natural language settings, like syntactic and lexical regularities and the interplay between them.

We started our investigation with 3 factors in mind: the number of negations in an expression, the logical operators used (AND or OR), and the truth value of the entire expression. But the results suggested that there are many more factors. It is reasonable to think that even more are waiting to be discovered. This research should therefore be expanded in the future to include the exploration of additional factors.

Concrete issues that beg further study include investigating situations involving expressions with different combinations of AND and OR, the possible effect of programming experience on understanding different types of expressions, and expressions based on computing conditions (e.g. `x <= 3`) rather than on given variables that are either TRUE or FALSE — and also whether there is a difference between `x < = 3` and `not(x > 3)`. Another whole line of research is the effect of short circuits. Developers undoubtedly exploit the possibility of short-circuiting in their work. It is therefore interesting to identify which patterns of logical expressions are more amenable to short circuits. In addition, it is also interesting to look into what developers do in practice, for example when they refactor logical expressions.

Another interesting issue is negations that appear in names rather than as logical operators. For example, consider the Boolean not_done when used in a loop header while (not_done) {...}. This can be compared with using a Boolean done with a negation operator as in the expression while (! done) {...} or alternatively a positive Boolean more_work and the expression while (more_work) {...}. We are working on such an experiment.

While we have only started studying these issues, we can already identify several possible practical implications for developers:

- We demonstrated that multiple negations have a detrimental effect on understanding. So negations should be avoided if possible. Examples: if you have a double negation, cancel them out. If an expression in an "if" can be flipped by removing a negation and switching the "then" and the "else", do so.
- Regularity also helps. So if an expression has repeated elements, try to emphasize its regularity.

Beyond these immediate implications, we hope that our findings will prompt additional research, which will eventually lead to a more comprehensive understanding of negations. By collecting many such results, we will be able to formulate more guidelines for developers that will help in steering towards more easily understood expressions.

EXPERIMENTAL MATERIALS

The experimental materials are available from Zenodo using the DOI 10.5281/zenodo.11064987.

ACKNOWLEDGMENTS

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

REFERENCES

[1] Galit Agmon, Yonatan Loewenstein, and Yosef Grodzinsky. 2022. Negative Sentences Exhibit a Sustained Effect in Delayed Verification Tasks. *J. Exp. Psy.: Learning, Memory, & Cognition* 48, 1 (Jan 2022), 122–141. https://doi.org/10.1037/xlm0001059

[2] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2019. Syntax, Predicates, Idioms — What Really Affects Code Complexity? *Empirical Software engineering* 24, 1 (Feb 2019), 287–328. https://doi.org/10.1007/s10664-018-9628-3

[3] Ruven E. Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. *Int. J. Man Mach. Stud.* 18, 6 (1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

[4] Bill Curtis, Jay Sappidi, and Jitendra Subramanyam. 2011. An evaluation of the internal quality of business applications: does size matter?. In *Proceedings of the 33rd International Conference on Software Engineering*,, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 711–715. https://doi.org/10.1145/1985793.1985893

[5] Viviane Déprez and M. Teresa Espinal (Eds.). 2020. *The Oxford Handbook of Negation.* Oxford University Press.

[6] I Deschamps, G Agmon, Y Loewenstein, and Y Grodzinsky. 2015. The processing of polar quantifers, and numerosity perception cognition. *Int. J. Man Mach. Stud.* 143 (2015), 115–128. https://doi.org/10.1016/j.cogni

[7] Dror G. Feitelson. 2022. Considerations and Pitfalls for Reducing Threats to the Validity of Controlled Experiments on Code Comprehension. *Empirical Software engineering* 27, 6, Article 123 (Nov 2022). https://doi.org/10.1007/s10664-022-10160-3

[8] Yosef Grodzinsky et al. 2020. Logical negation mapped onto the brain. *Brain Structure and Function* 35 (2020), 19–31. ttps://link.springer.com/article/10.1007/s00429-019-01975-w

[9] Yosef Grodzinsky et al. 2021. A linguistic complexity pattern that defies aging: The processing of multiple negations. *Journal of Neurolinguistics* 58 (2021), 543–554. https://www.sciencedirect.com/science/article/pii/S0911604420301421

[10] Sallie M. Henry and Dennis G. Kafura. 1981. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering* 7, 5 (1981), 510–518. https://doi.org/10.1109/TSE.1981.231113

[11] Errol R. Iselin. 1988. Conditional Statements, Looping Constructs, and Program Comprehension: An Experimental Study. *Intl. J. Man-Machine Studies* 28, 1 (Jan 1988), 45–66. https://doi.org/10.1016/S0020-7373(88)80052-X

[12] Marcel Adam Just and Patricia Ann Carpenter. 1971. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior* 10 (1971), 244–253. https://www.sciencedirect.com/science/article/abs/pii/S0022537171800518

[13] Sangeet Khemlani, Isabel Orenes, and P.N. Johnson-Laird. 2014. The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica* 151 (2014), 1–7. https://www.sciencedirect.com/science/article/abs/pii/S0001691814001206

[14] Robert C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftmanship.* Prentice Hall.

[15] Glenford J. Myers. 1977. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notices* 12, 10 (1977), 61–64. https://doi.org/10.1145/954627.954633

[16] Isabel Orenes, Linda Moxey, Christoph Scheepers, and Carlos Santamaría. 2016. Negation in context: Evidence from the visual world paradigm. *Quarterly Journal of Experimental Psychology* 69 (2016). https://doi.org/10.1080/17470218.2015.1063675

[17] Václav Rajlich and George S. Cowan. 1997. Towards Standard for Experiments in Program Comprehension. In *5th International Workshop on Program Comprehension.* 160–161. https://doi.org/10.1109/WPC.1997.601284

[18] Margaret-Anne D. Storey. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proc. 13th International Workshop on Program Comprehension.* IEEE Computer Society, 181–191. https://doi.org/10.1109/WPC.2005.38

[19] Ye Tian and Richard Breheny. 2015. Dynamic Pragmatic View of Negation Processing. *Negation and Polarity: Experimental Perspectives* 1 (2015), 21–43. https://link.springer.com/chapter/10.1007/978-3-319-17464-8_2

[20] P. C. Wason. 1959. The Processing of Positive and Negative Information. *Quarterly Journal of Experimental Psychology* 11 (1959), 92–107. https://doi.org/10.1080/17470215908416296

2.2 Is "notDone" the Same as "!done"? The Effect of Different Ways for Expressing Negation

Published: Aviad Baron and Dror G. Feitelson. In ACM Conference on International Computing Education Research V.1 (ICER 2025 Vol. 1), August 3–6, 2025, Charlottesville, VA, USA. ACM.

In this paper, we set out to investigate the various manifestations of negation in the context of code, how different types of negation affect code comprehension, and the practical implications for writing readable code. Programming languages feature a wide range of negation forms—not only the logical negation operator, as explored in the previous paper, but also negation embedded in variable names, alternative operators expressing negation, and the truth values of expressions. In addition, we explore different strategies for avoiding negation. These include, for example, checking whether an array is non-empty using a greater-than operator, or using a variable name that is a synonym of a negatively phrased name.

We examined the cognitive load associated with each type of negation, the interactions between them, and how the diversity of negation forms—including both syntactic negation (e.g., logical operators) and semantic negation (e.g., natural language negation in variable names)—impacts code comprehension. This research yielded insights into best practices for writing readable code, including which operators are preferable and how variable names should be formulated.

Is “notDone” the Same as “!done”? The Effect of Different Ways for Expressing Negation

Aviad Baron
The Hebrew University
Jerusalem, Israel
aviad.baron@mail.huji.ac.il

Dror G. Feitelson
The Hebrew University
Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Negation has been studied extensively in the fields of linguistics, psychology, and logic. However, it has been almost entirely overlooked in the realm of code comprehension research and the teaching of programming. Negations in code are interesting for several reasons. First, negations can be expressed either using logic operators (like ! or !=) or else by words embedded in variable names (as in notDone). Second, different types of negations can be combined together in the same expression. To explore whether using different negative expressions affects code comprehension, we conducted a controlled experiment involving 268 participants. The task was to understand short code snippets containing various logical expressions and types of negations. The results showed significant differences between the comprehension of different code snippets, both in terms of time needed and in terms of the correctness achieved. This illustrates a cognitive complexity that has important implications for writing more readable code and for guiding refactoring practices. In particular, we suggest that students be taught to avoid negations if possible, e.g. by using len > 0 rather than len != 0 to verify that an array is not empty.

CCS Concepts

• **General and reference** → **Design**; *Experimentation*; • **Theory of computation** → *Programming logic.*

Keywords

Code comprehension, Logical expression, Negation

ACM Reference Format:
Aviad Baron and Dror G. Feitelson. 2025. Is “notDone” the Same as “!done”? The Effect of Different Ways for Expressing Negation. In *ACM Conference on International Computing Education Research V.1 (ICER 2025 Vol. 1), August 03–06, 2025, Charlottesville, VA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3702652.3744213

1 Introduction

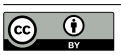
Negation is a fundamental feature of human language. As linguist Larry Horn writes [12], “In many ways, negation is what makes us human, imbuing us with the capacity to deny, to contradict, to misrepresent, to lie, and to convey irony.” The processing of sentences with and without negation, including instances of double negation and various logical relations, has been the focus of extensive research. Studies have also utilized tools such as fMRI to map the processing of logic to specific brain regions, offering insights into how the human mind navigates and comprehends intricate logical structures [1, 7, 10, 11, 15, 16, 26, 27].

Program comprehension is a critical cognitive process in software development, as developers dedicate a substantial portion of their time to understand existing source code [18, 28]. This process has a significant impact on maintenance and refactoring, as developers construct mental models that represent the code’s structure and functionality [5, 24]. It is also exceedingly important when canvassing code suggested by generative language models. The less code developers write themselves, the more important it is that they read and understand the code that is generated automatically.

The insights gained from research on understanding logical expressions and negation in natural language do not necessarily extend to the comprehension of code. Programming languages utilize precise mathematical notation that differs significantly from the nuances of natural language. As a result, issues such as the scope of negation, which are important in linguistic contexts, become irrelevant in programming. Moreover, the formal syntax and semantics of code allow for the construction of more complex expressions, such as lengthy formulas involving multiple variables and logical operators, which have no direct counterparts in natural language. In addition, programmers typically possess a strong background in logic and mathematics, making them an unrepresentative sample of the general population.

Negation in code, including the embedding of negation in variables names, is a challenge faced by many developers (e.g. [8, 9, 14]). However, the pedagogical literature rarely addresses the question of how to write Boolean conditions in a readable manner, particularly regarding the definition of Boolean conditions with negation and the choice among different logically-equivalent formulations. For example, *The Pragmatic Programmer* [25], a widely acclaimed book on the mastery of programming, does not address the question of how to write readable Boolean expressions with negation at all. *Clean Code* [17], another well-known programming handbook, mentions it only briefly in a single example (on p. 302), saying just that negations should be avoided.

In computing education research too there has been essentially no work on what makes Boolean expressions hard or easy to understand. Stefik and Siebert’s classic study on the intuitiveness of programming language constructs included some elements of such expressions [23]. While they did not include the negation operator, they did show that non-programmers considered `unequal` to be the most intuitive way to express the notion of inequality between operands, whereas programmers preferred the `!=` notation. As their



ICER 2025 Vol. 1, Charlottesville, VA, USA

ACM ISBN 979-8-4007-1340-8/25/08
https://doi.org/10.1145/3702652.3744213

focus was on individual language elements, they did not investigate alternatives for actually forming Boolean expressions.

Likewise, research on the comprehension of logical expressions—and particularly those involving negation—remains surprisingly sparse in the software engineering and code comprehension literature. Early work by Iselin considered loop conditions with "equals" and "not equal" operators (in Cobol) [13]. Ajami et al. conducted a study comparing three expression that use negation with a similar condition without negations [2]. Their findings revealed a significant difference in the time required to understand the code between two of the negative forms—specifically, a De Morgan pair—and the third negative form. However, the study did not provide an explanation for this observed difference. Baron et al. investigated the comprehension of negation in Python, focusing primarily on the logical `not` operator [4]. Their findings confirmed that negation is indeed more challenging to process. Additionally, they identified two significant factors: "syntactic regularity" and "logical regularity". They argued that expressions are easier to understand when all variables have negations or none do, and when all literals have the same truth value (either `true` or `false`).

Against this backdrop, we perform an exploratory study of the relative understandability of various forms of negation that are used in code. For example, we compare the loop condition `while(!done)`, which uses a logic operator, with the condition `while(notDone)`, which expresses exactly the same idea but embeds it in a variable name. In addition, we extend the scope to using various different operators. For example, checking that an array is not empty can be done by comparing its length to zero using `!=`, and also using `>`, which does not involve a negation. As far as we know, such variations have never been investigated before.

Note that the differences between these expressions are expected to be extremely subtle. It is not clear in advance that they are even measurable. For example, one might think that any experienced programmer learns to interpret and read `!` as `not`, and therefore there will be no effect. But from a cognitive point of view, even if developers may learn to interpret the different expressions correctly, they may be employing different parts of their brains to do so: an operator may be processed in the part of the brain that deals with arithmetic and logic, while a word may engage the language center. It stands to reason that the performance of these two paths will not be identical.

We therefore needed to design an experiment that isolates these differences, and perform it with enough participants to observe the differences that may be present. We collected data from 268 professional developers from around the world, nearly two thirds of them with more than 2 years of experience. Together, they provided 3052 individual measurements. These results enabled us to identify various differences that exist in the understanding of similar Boolean expressions, both in terms of achieving correctness and in terms of the time needed to do so.

The experiment comprised three parts. The first part compares logical expressions containing different operators: 'greater than', 'equal to' with negation, and 'not equal to'. In the second part, we examine different types of negation and the interactions between them, particularly focusing on variable names that include negation. The third part of the experiment explores variable names with and without negation in the context of a `while` loop. In total this led to the use of 42 short code snippets. Our participants were tasked with determining the output that would be printed by each such code snippet.

This study has the potential to offer practical guidelines for writing code, as well as cognitive insights into how negation is represented in the coding world, mapping these expressions and understanding the interactions between them. Our contributions in this paper are:

- Extending existing research on negation in natural language to a completely new context.
- Identification of different types of negations that reflect the coding world. These include
  - Explicit Logical negation: the `!` operator
  - Operators that contain an embedded negation: the `!=` operator
  - Names that contain negation, as in `notDone`
  - A feeling of negativity, as in `empty` or when a condition has a value of `false`.
- Comparison of equivalent ways to write a negative expression and their relative processing difficulty.
- Establishing the effect of interactions between these negations, like a double negation involving a variable name and operator together.
- Guidelines for writing readable variables names and Boolean expressions, for novice students in computer science courses as well as practitioners.

2 Research Questions

Our experiment focuses on comparing the understanding of short code snippets that involve various logical expressions with negation. We started by identifying the different types of negation that may occur in code. The first is obviously the logical negation operator itself, written as `!` in languages like C and JavaScript. But there are many other ways to express the notion of negativity, like the 'not equals' operator `!=`, and even using "not" in a variable name. All these different forms of negativity can complicate code comprehension. The research questions concern the significance of these different forms and possible interactions between them.

In this context, our research questions are as follows. The first concerns different negation operators:

(RQ1) *What is the effect of different negation operators?* This question can be divided into two sub-questions:
  (RQ1a) What is the effect of using a negative operator? Here we want to compare expressions with negative operators, namely `!` and `!=`, with expressions that do not contain them.
  (RQ1b) What is the difference between different negation operators? Here the focus is on equivalent ways to express the same logic: for example, either using `!=` or alternatively applying `!` to an expression with `==`.

The next question extends the discussion to include negativity in variable names.

(RQ2) *Do negated variable names make the code more difficult to understand?* This introduces the human element: we expect that a human developer may be affected by the word "no"

embedded in a variable name, even though formally there is no logical negation present.

Another issue is the effect of the construct that provides the context for the logical expression:

(RQ3) What are the interactions between different types of negation and conditions in a `while` loop?

In addition, there are some general crosscutting questions:

(RQ4) *What are the interactions between different types of negation?* Is double negation more difficult to process? Are code snippets with multiple instances of negation harder to understand?

(RQ5) *Does a condition that evaluates to* `false` *make the code more difficult to understand?*

(RQ6) *Does using words with a negative connotation have an effect?* Natural language allows us to express a negative notion without using negation explicitly, for example “empty” or “fail”. Does using such words have a similar effect to using explicit negations?

Finally, an important practical question is: *What are the implications of all the above for code writing?*

In our present exploration of all these questions, comprehension is defined as finding what a code snippet prints, and difficulty is measured by the time this took and the fraction of wrong answers. The questions of whether the results depend on these choices are left for future work.

3 Experimental Design and Execution

The experiment included five groups of code snippets, with 4 to 12 snippets in each. Two of the groups included simple expressions, two had `if` statements, and one had `while` loops. Each participant was given 14 code snippets selected randomly to represent the different groups and sub-groups. These snippets were ordered randomly. In addition all participants were given an introductory snippet as explained below. For all snippets, participants were asked to determine what the code would print. The code snippets were written in JavaScript, which is currently the most popular programming language according to the Stack Overflow developer survey[1]. The experiment was approved by the faculty ethics committee.

3.1 Experiment Design

The present paper focuses on simple expressions and on logical conditions in `while` loops, so we only describe these parts of the experiment.

The *first part* focused on basic operators, particularly negation, in order to answer questions RQ1 and RQ5. Each code snippet contains a basic expression, with the following code structure. The first line is the initialization of an array, either with fruits or as an empty array. The second line prints the result of some test on the array’s size. This is expressed in different ways, comparing the length of the array to 0 with or without using negation:

- `fruits.length != 0`
- `!(fruits.length == 0)`
- `fruits.length > 0`
- `fruits.length == 0`

[1] https://survey.stackoverflow.co/2024/technology

Each of these options is applied to an empty array or to an array with some contents, for a total of 8 snippets. For instance, in the following code, the array is not empty and the expression is `fruits.length > 0`.

```
let fruits = ["Cherry", "Pear", "Apple"];
console.log(fruits.length > 0);
```

As another example, in the following code the array is empty and the expression is `!(fruits.length == 0)`.

```
let fruits = [];
console.log(!(fruits.length == 0));
```

The *second part* of the experiment contained code snippets with Boolean variables, designed to answer Research Questions RQ2, RQ4 and RQ6. Some of the names included an embedded “no”, thereby introducing an apparent semantic negation but without an explicit negation operator. We aimed to examine the effect of such names relative to other negations. As in the previous part, the code snippets contained an array of fruits, either full or empty. The variable names used to describe the array were:

- a positive name, `hasFruit`;
- a name with explicit negation, `hasNoFruit`; and
- a name with the same meaning but without explicit negation, `isEmpty`.

In each snippet the variable was initialized according to its meaning (so names were not misleading). It was then printed with or without a logical negation operator. One example is the following snippet. The array is empty, the variable is named `hasNoFruit`, and the condition includes negation:

```
let fruits = [];
let hasNoFruit = fruits.length == 0;
console.log(!hasNoFruit);
```

As another example, in the following code the array is full, and the variable `isEmpty` is printed without negation:

```
let fruits = ["Cherry", "Pear", "Apple"];
let isEmpty = fruits.length == 0;
console.log(isEmpty);
```

In total, in this part we have 12 code snippets: two possibilities for whether the array is full or empty, multiplied by three names for the Boolean variables, multiplied by two possibilities for whether there is a negation operator in the expression.

In the *third part* we examine the interaction between variable names with and without negations and understanding the conditions in a `while` loop, in order to answer RQ3. There were two sets of 3 snippets each. In each set there were 3 names, one positive and two negative, with the negation either embedded in the name or using the negation operator. The loop control statements in the first set were:

- `while (hasMoreWork)`
- `while (notDone)`
- `while (!done)`

The loops iterated over an array with numbers, and the control variables were initialized accordingly. Below is one of the code snippets from this set:

```
let numbers = [1, 2, 3];
let hasMoreWork = numbers.length > 0;
while (hasMoreWork){
    console.log(numbers[numbers.length -1]);
    numbers.pop();
    hasMoreWork = numbers.length > 0;
}
```

In the second set, the code snippets represent a scenario of using a buffer. The three names used were `hasSpace` and `notFull`, which have the same meaning, one with a negation and the other without, and `full`, which has the opposite meaning and is accompanied by a negation operator in the loop condition. An example of one of the code snippets is:

```
let numbers [];
let notFull = true;
while (notFull){
    if (numbers.length < 3){
        numbers.push(1);
    }else{
        notFull = false;
    }
}
console.log(numbers);
```

In the different parts of the experiment we deliberately included variable names with "no" in the middle, like `hasNoFruit`, and names with "not" at the beginning, such as `notDone`. The variable name with "no" was used in the second part of the experiment to investigate the case of double negation. The idea was to avoid having two negations in close proximity, to ensure the negation calculation is done in two separate steps and is not quickly canceled out due to the proximity of the negations (as would happen in an expression like `!notDone`).

The full experiment also included codes about misleading names and using `if`s, which are not included in this paper.

3.2 Methodological Considerations

The programming language chosen for the experiment is JavaScript, for several reasons. Firstly, it is a widely popular language that many developers are familiar with, as mentioned above. Another reason is that negation is expressed in JavaScript using `!` rather than the word "not". This is similar to other important languages like Java and C/C++. It also offers an advantage because, unlike in Python, the expression is more distanced from natural language. This allows for a distinction between logical negation using `!` and negation in variable names using "not" or "no". An experiment in Python would not capture the distinct differences between types of negation as clearly.

One of the main methodological consideration was to create the most atomic code snippets possible. This ensures that no additional elements were included beyond what was necessary to examine the factors in the research questions. This approach aimed to keep the experiment as clean as possible, minimizing any threats to validity arising from confounding influences that could affect the results.

It is sometimes observed that the first question in an experiment takes more time to answer, as participants need to get accustomed to the environment and what is required of them (e.g. [2, 22]). Additionally, we were concerned that there might be a slight bias due to naming the array "fruits" if the first randomly-chosen snippet actually had an empty array, so the code in fact did not contain any mention of fruits. To avoid these problems and provide training, a generic initial code snippet not related to the experiment was included for everyone. This snippet differed from the other code snippets so as not to provide any priming for any of them. Nonetheless, it did include the initialization of an array with fruits to ensure the context was clear. The chosen initial code snippet was:

```
let fruits = ["Cherry", "Pear", "Apple"];
let hasApple = false;
for (let fruit of fruits){
    if (fruit == 'Apple')
        hasApple = true;
    }
}
console.log(hasApple);
```

Above the code snippet was a reminder about the `console.log` command in JavaScript, which is the common method for outputting information (since a `print` function does not exist). In addition, we had a reminder for `pop` just before the participant received a question involving this command. These reminders are expected to help participants who may not be versed in JavaScript, but can still contribute to the experiment because the experiment is focused on basic expression and does not really depend on specific features of the language.

In the questions of parts 1 and 2, due to their brevity and the clear binary nature of the answer (true or false through the evaluation of an expression), we methodologically preferred to structure them as multiple-choice questions. This is because even typing out "true" or "false" takes some time, and given that the total time is very brief, we aimed to prevent any influence from the time taken to write the response. Therefore, we opted for a format where the answer could be selected with a single click in a multiple-choice question format. In part 3 (the `while` questions) there are many possible wrong answers, so it is not practical to use a multiple-choice format.

Only a subset of code snippets (14 in total) was selected for each participant, to reduce the length of the experiment and reduce fatigue and attrition. Having fewer questions may also prevent settling into a more technical and routine reading, that might not accurately reflect a general understanding of the code snippets. The selection method ensured that the snippets were distributed across different groups, leading to a greater variety of code examples and minimizing the impact of technical and focused reading due to structural similarities within groups.

Additionally, the order of code snippets was randomized to prevent any systematic bias in the results that could arise from the sequencing of the code snippets rather than their difficulty.

3.3 Experiment Execution

The experiment started with an introductory page explaining what the experiment is about. This included details about the number of questions, the approximate time the experiment is expected to take, and a general overview of the experiment's purpose: "Our

goal is to understand the cognitive mechanisms of reading different logic patterns in codes”. Additionally, participants were informed that the experiment involves measuring response times, and they were instructed to respond only when fully focused and without any distractions.

The experiment was conducted using the Qualtrics platform. Qualtrics supports measuring response times, with the key metric being the total seconds the question was visible before the respondent clicked for the last time.

A total of 268 participants were recruited using programming-related channels (e.g. in Reddit). Among those who reported their educational background, 78 held a Bachelor’s degree, 28 held a Master’s degree, and 6 held a Ph.D. Additionally, 55 participants were self-taught, had received vocational training, or learned to program in high school. Regarding professional experience, among those who provided this information, 61 participants reported having 0-2 years of experience, 70 participants had above 2 up to 6 years, and 33 participants had more than 6 years of experience. In terms of gender, among those who reported their gender, the participant group was predominantly male, with 150 male participants, 8 female participants, and 1 non-binary/third-gender participant.

4 Results

Our experiment included multiple code snippets in three parts, and many comparisons were made in the process of analyzing the results. However, we note that these actually relate to individual questions, which are each of interest independently. For example, the comparison of != with ! applied to == is unrelated to the comparison of names with and without negations in loop conditions. In such situations a correction for multiple experiments is not required [3, 20].

To mitigate any remaining concerns, an additional viewpoint is as follows. In the following we report 25 p-values, 14 of which are smaller than 0.05. With this number of comparisons and this threshold one would expect 0-2 false positives, not 14. Considering each comparison as a Bernoulli trial with p=0.05, the probability of 14 successes in 25 trials is 0.0000000000015. If you have 1 or 2 positive results they indeed may reflect chance. If you have 14 the vast majority must be true.

4.1 Part 1: Operators Expressing Negations

Recall that in this part, we are examining simple logical expressions. The comparison is between the expressions:

- `fruits.length != 0`
- `!(fruits.length == 0)`
- `fruits.length > 0`
- `fruits.length == 0`

For each code snippet, we have two results: the fraction of participants who understood it correctly, and the time they took to do so. Figure 1 shows the CDFs (cumulative distribution functions) of the time taken. The time is on the horizontal axis, and the graph shows the probability of solving the problem within a given time. Therefore, a line positioned further to the right indicates a longer duration needed for a correct response. The graphs account for correctness by assigning an infinite time to incorrect answers. As a result, the CDFs do not achieve a maximum value of 1, but instead converge to the fraction of correct answers. When the code snippets are very basic, participants almost always responded correctly, making this fraction 1 or nearly 1. But in some cases there are sizable differences in correctness, as can be seen by the gap between the right-end of the plot and 1.

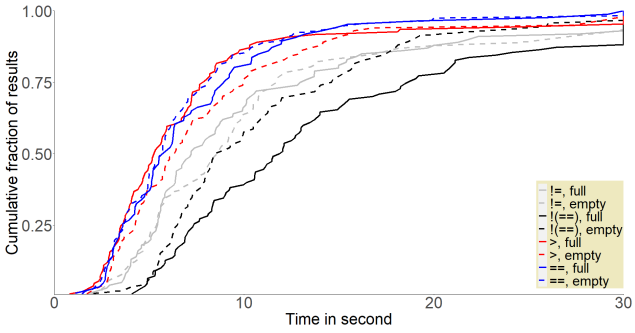


Figure 1: CDFs of the time to correct answers for logical expressions of part 1.

Figure 1 shows an overview of all the results of this part of the experiment together. Generally, significant differences can be observed between the various conditions. The two versions that were the fastest to understand were when the array is full and the condition checks if it is greater than zero, and when the array is empty and the condition checks if its size equals zero. Conversely, conditions involving negation are less readable, with the least readable case being those using the logical negation operator `!(fruits.length == 0)`.

To uncover the factors that lead to these results we analyze the effect of each one separately.

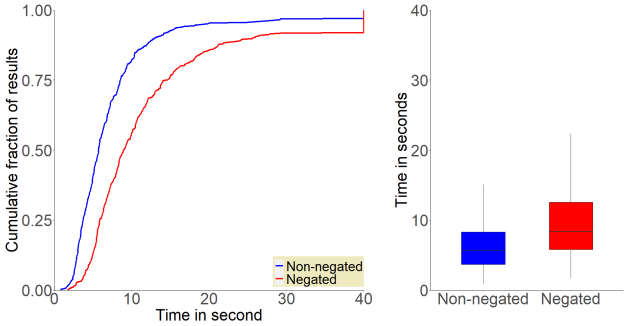


Figure 2: CDFs and boxplots of time to correct answers for negated vs. non-negated logical expressions. The CDFs include incorrect answers as ∞, the boxplots do not.

4.1.1 The Effect of Negativity. The first factor we considered, in order to examine Research Question RQ1, is the effect of negative operators. Our code snippets can be partitioned into two groups of 4 by their negativity. The “negative” group includes the expressions with a “!” in them: either using the operator != or directly negating

the 'equals' operator. The "non-negative" group uses > or == with no such negations. Figure 2 compares the combined results of these two groups. In addition to the CDFs which incorporate the correctness results as explained above, it also shows boxplots of the time needed for correct answers only. This shows that the negative expressions take longer to process than the non-negative ones, and also lead to more errors. We conclude that the negations may induce some burden which makes these expressions harder to understand than expressions without negations.

More formally, the independent variable has two levels, representing the negative and positive groups. The dependent variable is the time of responses. The comparison is between subjects. We would like to check whether the average time needed to process the different versions (in pairs) is equal. For this we will use 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 these tests is that there is a statistically significant difference between negative and non-negative expressions ($p < .0001$), and the effect size, as measured by Cohen's d, is 0.66 indicating a medium effect. Thus the null hypothesis was rejected. In addition, we would like to check whether the expected values of the percentage of errors made in different situations are unequal. For this we will use a between-subjects Z-test for proportions. The null hypothesis is that the expected percentage of the participants who answer correctly is equal in both cases, and the alternative hypothesis is that the expected values are different. According to the statistical test this difference too is significant ($p = .002$), and the effect size, as measured by Cohen's h, is 0.22 indicating a small effect.

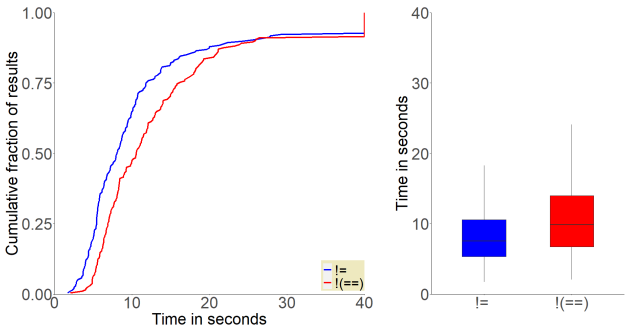


Figure 3: CDFs and boxplots of of the time to correct answers for logical expressions with != vs. !(==).

4.1.2 Expression of Negativity. In the previous subsection we bundled two forms of negativity together: The use of the 'not equal' operator `!=`, and the construct where an explicit logical negation `!` is applied to the 'equals' operator == [in the sequel we refer to this construct as `!(==)` for brevity]. We now compare these two forms to each other, which is also part of Research Question RQ1. The results, shown in Figure 3, indicate that `!=` is more readable than `!(==)`. Applying the t-test as previously showed that the difference is statistically significant ($p < .0001$), and the effect size, as measured by Cohen's d, was 0.41 indicating a small effect. But there was no significant difference in the ratio of correct answers out of the total responses ($p = .619$).

We explain this by noting that the processing of `!(==)` involves two stages. For instance, consider the expression

```
!(fruits.length==0)
```

It includes the first processing stage of finding the truth value of `fruits.length==0`, followed by the second stage of applying the negation. In contrast, the expression using `!=`, which also means "not equal to", operates in a single step. This interpretation leads us to the following analysis, which reinforces the explanation provided here.

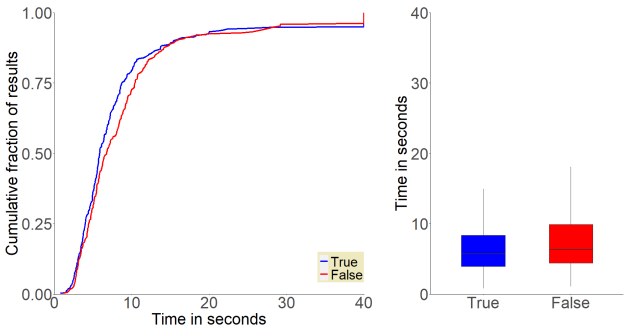


Figure 4: CDFs and boxplots of the time to correct answers for logical expressions with different truth values.

4.1.3 Effect of the Truth Value. Another possible factor we considered is the truth value itself, in order to examine Research Question RQ5 — maybe there is some cognitive hindrance attached to dealing with falsehood? To look into this we focus on 3 of our 4 expressions, excluding the one with an explicit negation of the 'equals' operator. The reason for doing so is that this expression contains two steps of calculating the truth value: the 'equals' and then its negation. By construction these two steps have opposite truth values. Therefore it is not possible to assign instances of this expression to either `true` or `false`.

Figure 4 shows that conditions with a `false` truth value are less readable than those with a `true` truth value. Thus the truth value of the statement does seem to have an impact. This observation is also supported by the statistical test, which indicated significance ($p = .0068$). However, the effect-size as measured by Cohen's d is 0.23 (small), and there is no difference between them in terms of the number of correct answers.

In interpreting this result, we do not necessarily claim that "truth is easier than falsehood". But we do note that truth inherently corresponds to reality. In our case, the code the experiment participants see is very short, and contains two elements: the initialization of an array and an expression describing this array. If the expression actually describes the array this immediately "pops out"—for example when the array is initialized as empty and the expression says its length equals 0. But when the expression does not correspond to the array initialization—for example when the array is initialized to empty but the expression says its length is greater than 0—this may

lead to some cognitive dissonance, and hence to a slightly longer processing time.

This insight can be used to also explain the difference between the two versions of the expression we excluded previously. In Figure 1 we can see that the expression `!(==)` with a `true` value takes more time to process than the same expression with a `false` value. The reason may be that when the complete expression evaluates to `false` the inner expression evaluates to `true`—namely, the expression indeed describes the array initialization in the previous line, which is faster to see.

4.2 Part 2: Negativity in Variable Names

Recall that in this section, we are examining logical expressions involving Boolean variables with the names `hasFruit`, `hasNoFruit`, and `isEmpty`. We evaluate these names under conditions where there is a logical negation or where there is none, and in situations where the condition evaluates to `true` or to `false`.

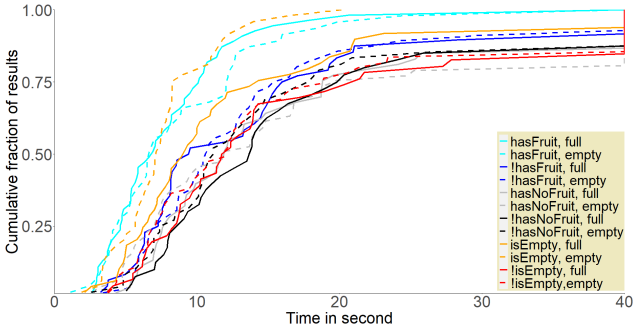


Figure 5: CDFs of the time for comprehending the 12 snippets in part 2.

Figure 5 presents an overview of all the results for this section. Again it can be observed that there are significant differences between the code snippets. The most obvious differences are that the light-colored lines, representing positive snippets, are above the dark ones, representing negativity. For example, the most readable code was when the array is full, the variable name is `hasFruit`, and there is no negation operator. In contrast, when the array is full, the variable is `hasNoFruit`, and there is a negation operator in the print statement, the expression took the most time to understand. We now turn to analyze the different effects one by one.

4.2.1 Negative Names. As noted above, we considered two alternative ways to express negativity in names, to answer Research Questions RQ2 and RQ6. The first is explicit negativity, as in the name `hasNoFruit`. The second is semantic negativity, as in the equivalent name `isEmpty`. We compare them with the positive name `hasFruit`.

We compare the combined results for each of these three names in Figure 6. We can see that the most readable name, in terms of both time and correctness, is `hasFruit`. According to the boxplots `isEmpty` has a very similar time distribution, but it does suffer from slightly more errors. However, the difference is not statistically significant (p = .396 overall and p = .229 for correct answers). The least readable name is the one with explicit negation, `hasNoFruit`.

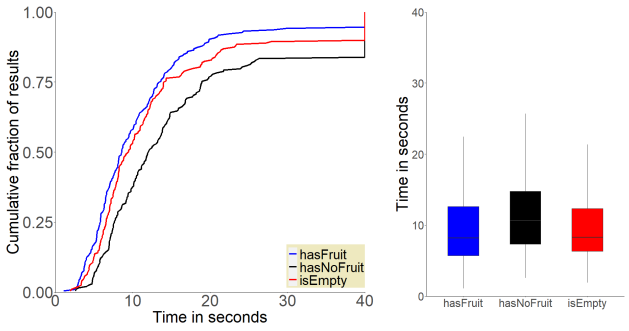


Figure 6: CDFs and boxplots of the time to correct answers for logical expressions with different variables names.

And the differences between this name and the previous two are indeed statistically significant (p = .0002 and p = .0044). The effect sizes, as measured by Cohen’s d, were 0.39 and 0.31, respectively, indicating a small effect. In addition, the first is also statistically significant for correctness (p = .0022) and the effect size is 0.57 (medium), while the second is not (p = .055).

Our conclusion is that a name with explicit negation like `hasNoFruit` may place a burden on processing. Therefore, instead of using a name with explicit negation (`hasNoFruit`), it is preferable to choose a synonym without negation(`isEmpty`), which may ease processing, or use the opposite non-negated option `hasFruit`. While we did not find a statistically significant difference between using the positive `hasFruit` and the semantically negative `isEmpty`, the separation of the CDFs in the graph indicates that the positive name might suffer from fewer errors. This is an interesting lead as there are also other cases of such negativity in connotation, such as failure versus success. But establishing whether they indeed have an effect requires further investigation.

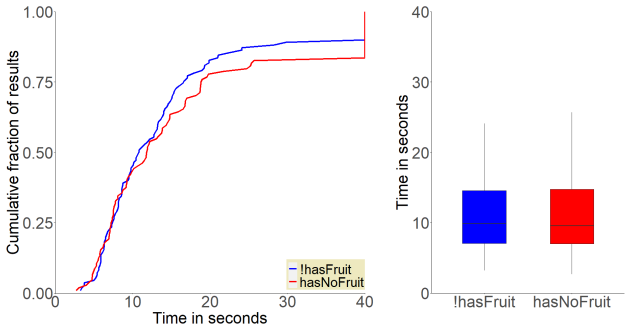


Figure 7: CDFs and boxplots of the time to correct answers for negated naming versus the negation operator.

4.2.2 Negated Name vs. Negation Operator. Returning to examine Research Question RQ1, considering the possibility of negation in names exposes two alternatives to express the exact same logic: either use the word “no” or the operator `!`. In our experiment this is represented by the pairs of code snippets where the expression

being printed is either `hasNoFruit` or `!hasFruit`. We compare them directly in Figure 7. The result is that overall there is no statistically significant difference (p = .847), and indeed, the boxplots show that the distributions of time to correct answer are practically identical. But `!` does have a statistically significant advantage in terms of the ratio of correct answers to total responses (p = .043) and the effect size is 0.26 (small). Namely, a negated variable name results in somewhat more errors compared to a positive name with a logical negation.

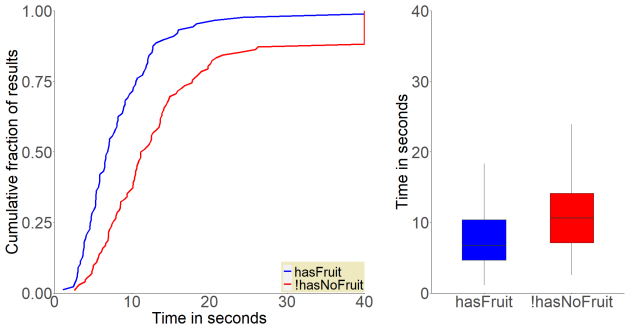


Figure 8: CDFs and boxplots of the time to correct answers for a double negated vs. non-negated expression.

4.2.3 Negated Names and Double negation. We showed above that names with negation, for example containing the word "no", take longer to process. But they may also interact with logical negation expressed using the `!` operator.

We first analyze this case by focusing on code snippets with two logically-equivalent expressions in the print statement in order to examine Research Question RQ4: `hasFruit` and `!hasNoFruit`. The results are shown in Figure 8. Obviously the expression with the `!hasNoFruit` took significantly longer (p < .0001, and an effect size of 0.83 (large)), and also caused many more incorrect answers (p = .011, with an effect size 0.51 (medium)). We contend that this reflects a "double negation", where one negation is a formal logical negation with the operator `!`, and the other is a semantic negation in the word "no". In other words, from the point of view of a human developer, variable names can lead to situations involving multiple negations, even if there is only one explicit negation in the logical expression.

The possibility of multiple negations of different types in the code leads us to counting their combined effect in the next subsection.

4.2.4 Multiple negations of Different Types. In the previous subsections we highlighted different types of negativity that may appear in the code. We also showed that they may combine to create "double negations", where the components are actually of different types. To further investigate this, we now consider all possible combinations together.

The significant components identified above are the following:

- the explicit logical negation of using the `!` operator;
- negation in the variable name, by embedding a "no" in it;
- the `false` truth value of a condition or assignment.

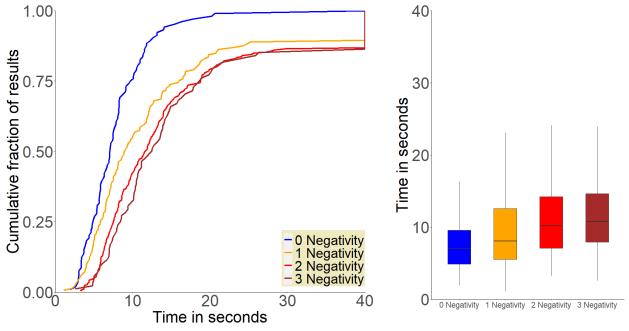


Figure 9: CDFs and boxplots of the time to correct answers for code snippets with different negativity counts.

We counted how many of these appear in each of the 12 code snippets, and group the snippets according to this number, from not having any negative component to having a maximum of 3 negative components. For example, the following snippet:

```
let fruits = [];
let hasFruit = fruits.length > 0;
console.log(!hasFruit);
```

is counted as 2: the expression initializing the variable is `false`, and the print statement contains the negation `!`.

Figure 9 presents the results. It indicates that the number of negations significantly affects the difficulty of understanding the code: the more negative components a piece of code contains, the longer it takes to process. In terms of statistical significance, the differences between 0 and 1 negative components and between 1 and 2 negative components are statistically significant (p = .0007 and p = .0058), and the effect size is small (Cohen's d of 0.43 and 0.23). The differences between 2 and 3 is not statistically significant (p = .328). In terms of error rate, the difference between 0 and 1 is statistically significant (p = .009), while the others are not.

4.3 Part 3: Negations in Controlling `while` Loops

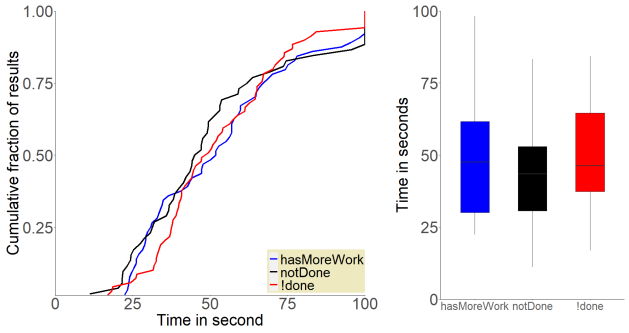


Figure 10: CDFs and boxplots of the time to correct answers for the first set of `while` snippets.

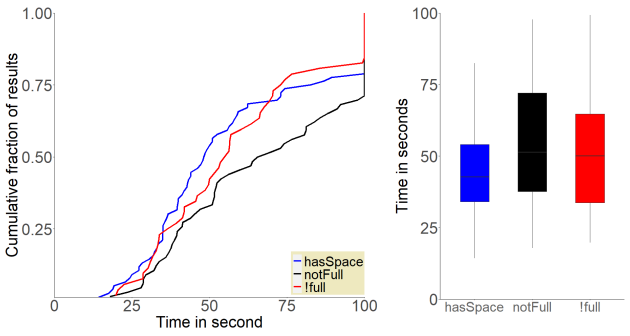


Figure 11: CDFs and boxplots of the time to correct answers for the second set of `while` snippets.

In this section, we examine the interaction between negation in variable names and logical negation within conditions in a `while` loop, to address Research Question RQ3. In Figure 10, we can see the results of the first set. Surprisingly, the name with the negation `notDone` was not less readable than the others, and actually there were no statistically significant differences at all (the three comparisons led to p = .105, p = .878, and p = .181). We discuss this below.

In Figure 11, we see the results of the second set. In this case it is evident that the name with the negation `notFull` is the least readable one—similar to the conclusions from the previous sections. And its difference from the positive name `hasSpace` is statistically significant (p = .018) and the effect size is 0.49 (small). The other differences between these conditions are not significant (`notFull` vs. `!full` p = .389, and `hasSpace` vs. `!full` p = .118).

Note the apparent disagreement between these two sets of results concerning negated names: in the first, `notDone` is quite readable, but in the second, `notFull` is the least readable. We believe the difference results from a unique interaction between the variable name `notDone` and the `while` loop construct: reading the condition `while (notDone)` spells out the essence of a generic loop where the execution of a block of code is repeated as long as the computation is not finished. It is true that `while (hasMoreWork)` is semantically equivalent; but the first expression is more succinct and in some sense seems more natural. This match between the program text and the semantics compensates for the presence of the negation in the variable name, leading to higher readability.

The names used in the second set represent the property of a buffer being full or not. Thus they do not relate to a generic loop structure, and we do not see the same effect. Therefore in this case the name with the negation turned out to be less readable, as was the case in the previous sections.

Another observation is that the results are not as clean as those in the previous sections. As noted above, only one difference was found to be statistically significant[2]. This indicates that there are probably more factors at play, and it is harder to isolate the effect of the way variable names are formed. The conclusion is that many more experiments with careful selection of the different treatments are needed.

[2]Note that in this part it is necessary to make a Bonferroni correction because the two sets of the `while` questions are parallel, and therefore there are two comparison that test each research question. However, the test that came out significant still remained significant after the correction.

5 Pedagogical implications for teaching code writing

In our study, we identified negation factors that contribute to the complexity of code. In this section, we aim to discuss the practical implications of our findings for writing more readable code. We observed that code may contain a wide range of negative expressions in various forms, all of which make processing more challenging. Therefore, as a general recommendation for writing code, it is usually advisable to avoid negations.

First, we found that the logical negation operator complicates processing, so when a condition can be expressed without this operator, it is generally preferable to do so. For instance, consider the case of checking that an array is not empty. The direct way to express this is using `!(array.length == 0)`. But we found that the alternative form `array.length > 0` is much more readable.

Additionally, using a variable name with a negation is generally not recommended, as such names are harder to process. Moreover, they can lead to even more difficult situations, such as double negation. For example, if we name a variable `isNotActive`, beyond the difficulty in processing the negation in the name, negating this variable would result in a situation like `!isNotActive`, which is even harder to process. Therefore, we recommend using a variable name without negation that conveys the same meaning (a synonym) or using the variable name without negation and simply initializing it differently.

Another observation we made is based on the comparison between the ‘not equals’ operator `!=` and the negation of the ‘equals’ oparator, where we found that the former is more readable. This led us to the insight that when there are different ways to express the same condition, it is important to consider the *number of computational steps* required as a factor in choosing the most effective expression. Expressions with more computational steps will probably be more difficult to understand.

6 Threats To Validity

Construct validity. We measured the difficulty of understanding different expressions by measuring time and checking the correctness of the answers. This faces two threats. Difficulty is not directly observable. We assume difficulty is reflected in time to solution and in correctness, which are commonly used proxies [19]. However, there have been indications that they are not always correlated, because correctness may also be impaired by unmet expectations [2]. We therefore measure both, and in our case they are indeed correlated in most of the cases, thereby providing a measure of support for each other. Note too that while assessing difficulty is interesting from a theoretical perspective, in practical terms the effects on time (and hence productivity) and correctness (bugs) are actually the important factors.

Internal validity. Many of our design decisions in formulating the code snippets for the experiment were taken to reduce threats to validity, as explained in Section 3.2. But it is possible that repeated exposure to short code snippets could alter reading patterns

in such a way that they become focused on certain specific elements, thereby reducing the influence of other relevant factors. For example, due to habituation, the presence of a variable name with a negation might no longer have an effect, as the variable name has already been seen, leading the eye to focus only on certain parts of the code without a thorough reading of the entire snippet. However, we believe this threat is not significant for two reasons. First, even a partial examination of the code is important, as it can indicate differences in the difficulty of understanding and the various contributing factors. Second, this concern is mitigated by the fact that we provided relatively few code snippets, some of which are entirely different—such as the `while` loop questions—thereby minimizing the potential for significant impact in this regard. In addition, because we randomized the presentation of the code snippets, any effect (if it exists) would be spread evenly across all the codes and there will be no systematic effect.

External validity. Research findings are always limited to the specific circumstances in which the research was conducted. For example, variable names could exhibit a much greater variety, and logical expressions might appear in a wider range of contexts. In our study, however, we sought to measure the most atomic conditions possible in order to isolate the specific factors under investigation. Thus we can not know whether the results generalize to more complex situations as may appear in real code. We also can not know whether similar results will be found in different populations, for example school children learning to program. Additional experiments and replication are always needed to establish the circumstances where results are valid.

7 Conclusions

Negation has been a focal point of research in natural language, as illustrated by the publications on the subject previously mentioned in the introduction. However, the context of code presents a novel and intriguing domain. In coding, there is a unique combination of different kinds of negations: natural language elements, such as variable names, alongside formal mathematical language, including logical operators. Moreover, these different types of negations may interact with each other. This makes code completely different from pure natural language in this respect.

The way code is read and understood differs significantly from that of reading and understanding prose [6, 21]. In addition, the developers who read code, with their formal mathematical training, most probably think differently about negations in code from laymen who use negations in natural language. Despite this, negation has been scarcely studied in the coding world, particularly regarding the relationships between various kinds of negations.

In this research, we aimed to characterize negation in the coding world: identifying the types of negation that exist and understanding their basic impact on code comprehension. Our study revealed that these negations significantly impact the processing of different code segments. Figure 9 illustrated the overall impact of combinations of different types of negations on the difficulty of code comprehension. Our findings indicate that even a single negation, regardless of its type, increases both the time required for comprehension and the likelihood of errors compared to code without any negations. Figure 8 depicts the interaction of a double negation, where a variable name is negated alongside the use of a negation operator. This interaction further complicates the understanding of the code, highlighting the challenges posed by multiple layers of negation.

Additionally, we observed that the differences in comprehension can be quite substantial. For instance, Figure 1 shows that even in relatively simple expressions, the impact of negation can be dramatic in terms of both the time taken and the number of errors made. For example, when checking whether an array is non-empty, the expression `!(length == 0)` was significantly less comprehensible than `length > 0`. This finding underscores the importance of the structure of the condition, as there are considerable disparities between seemingly simple expressions.

These insights emphasize the critical role that negation plays in code comprehension and highlight the need for careful consideration in how negations are expressed and structured within code. The significance of this research is twofold. First, it enhances our understanding of code comprehension, and provides empirical support for recommendations on writing logical expressions and Boolean variables in code, including pedagogical guidelines for teaching novice developers. Additionally, it expands the study of negation into a new and interesting context beyond its use in natural language.

Data Availability

All experimental materials and data are available on Zenodo using the DOI 10.5281/zenodo.14974143.

References

[1] Galit Agmon, Yonatan Loewenstein, and Yosef Grodzinsky. 2022. Negative Sentences Exhibit a Sustained Effect in Delayed Verification Tasks. *J. Exp. Psy.: Learning, Memory, & Cognition* 48, 1 (Jan 2022), 122–141. doi:10.1037/xlm0001059

[2] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2019. Syntax, Predicates, Idioms — What Really Affects Code Complexity? *Empirical Software engineering* 24, 1 (Feb 2019), 287–328. doi:10.1007/s10664-018-9628-3

[3] Richard A. Armstrong. 2014. When to use the Bonferroni correction. *Ophthalmic & Physiological Optics* 34 (2014), 502–508. doi:10.1111/opo.12131

[4] Aviad Baron, Ilai Granot, Ron Yosef, and Dror G. Feitelson. 2024. Understanding Logical Expressions with Negations: Its Complicated. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024*. ACM, 303–312. doi:10.1145/3661167.3661180

[5] Ruven E. Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. *Intl. J. Man-Machine Studies* 18, 6 (1983), 543–554. doi:10.1016/S0020-7373(83)80031-5

[6] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proc. 25th International Conference on Program Comprehension*. 255–265. doi:10.1109/ICPC.2015.36

[7] Viviane Déprez and M. Teresa Espinal (Eds.). 2020. *The Oxford Handbook of Negation*. Oxford University Press.

[8] IBM forum. 2019. Negative variable names. (2019). https://www.ibm.com/support/pages/negative-variable-names.

[9] Quora forum. 2018. Is it bad practice to have a negative boolean variable name like 'didntLose' instead of 'didLose'? (2018). https://www.quora.com/Is-it-bad-practice-to-have-a-negative-boolean-variable-name-like-didntLose-instead-of-didLose.

[10] Yosef Grodzinsky et al. 2020. Logical negation mapped onto the brain. *Brain Structure and Function* 35 (2020), 19–31. ttps://link.springer.com/article/10.1007/s00429-019-01975-w

[11] Laurence R. Horn. 1989. *A Natural History of Negation*. University of Chicago Press.

[12] Laurence R. Horn (Ed.). 2010. *The Expression of Negation*. De Gruyter Mouton.

[13] Errol R. Iselin. 1988. Conditional Statements, Looping Constructs, and Program Comprehension: An Experimental Study. *Intl. J. Man-Machine Studies* 28, 1 (Jan 1988), 45–66. doi:10.1016/S0020-7373(88)80052-X

[14] Dave Jachimiak. 2018. Avoid Negative Variable Names. (2018). https://davej.io/2018/05/negative-naming.html.
[15] Marcel Adam Just and Patricia Ann Carpenter. 1971. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior* 10 (1971), 244–253. https://www.sciencedirect.com/science/article/abs/pii/S0022537171800518
[16] Sangeet Khemlani, Isabel Orenes, and P.N. Johnson-Laird. 2014. The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica* 151 (2014), 1–7. https://www.sciencedirect.com/science/article/abs/pii/S0001691814001206
[17] Robert C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall.
[18] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proc. 23rd International Conference on Program Comprehension*. 25–35. doi:10.1109/ICPC.2015.12
[19] Václav Rajlich and George S. Cowan. 1997. Towards Standard for Experiments in Program Comprehension. In *5th International Workshop on Program Comprehension*. 160–161. doi:10.1109/WPC.1997.601284
[20] Mark Rubin. 2024. Inconsistent multiple testing corrections: The fallacy of using family-based error rates to make inferences about individual hypotheses. *Methods in Psychology* 10, Article 100140 (Nov 2024). doi:10.1016/j.metip.2024.100140
[21] Mor Shamy and Dror G. Feitelson. 2023. Identifying Lines and Interpreting Vertical Jumps in Eye Tracking Studies of Reading Text and Code. *ACM Trans. Applied Perception* 20, 2, Article 6 (Apr 2023). doi:10.1145/3579357
[22] Andreas Stefik and Ed Gellenbeck. 2011. Empirical Studies on Programming Language Stimuli. *Softw. Quality J.* 19, 1 (Mar 2011), 65–99. doi:10.1007/s11219-010-9106-7
[23] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programing Language Syntax. *ACM Trans. Computing Education* 13, 4, Article 19 (Nov 2013). doi:10.1145/2534973
[24] Margaret-Anne D. Storey. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proc. 13th International Workshop on Program Comprehension*. IEEE Computer Society, 181–191. doi:10.1109/WPC.2005.38
[25] David Thomas and Andrew Hunt. 2020. *The Pragmatic Programmer*. Pearson Education.
[26] Ye Tian and Richard Breheny. 2015. Dynamic Pragmatic View of Negation Processing. *Negation and Polarity: Experimental Perspectives* 1 (2015), 21–43. https://link.springer.com/chapter/10.1007/978-3-319-17464-8_2
[27] P. C. Wason. 1959. The Processing of Positive and Negative Information. *Quarterly Journal of Experimental Psychology* 11 (1959), 92–107. https://doi.org/10.1080/17470215908416296
[28] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 951–976. doi:10.1109/TSE.2017.2734091

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

Status: Under review

Aviad Baron and Dror G. Feitelson.

In this paper, we conducted an in-depth examination of the comprehension of logical conditions involving negation. Drawing on distinctions found in the literature regarding different levels of code understanding, we identified two levels of comprehension relevant to the specific context of Boolean conditions: "tracing" and "comprehension". We then developed a tailored methodology to assess both levels of understanding. The methodology is based on the distinction between understanding how the code behaves for a single input, and a more advanced level of comprehension generalizing its behavior across all possible inputs within the given context.

We identified several factors that influence the difficulty of interpreting Boolean conditions—some of which affect only one level of comprehension. Additionally, we examined the understanding of logically equivalent conditions and explained discrepancies in comprehension through the cognitive factors we identified. Based on these findings, we offered practical guidelines for writing more readable Boolean expressions by choosing among logically equivalent alternatives.

A key insight from this study is that negation increases the difficulty of understanding logical conditions that involve disjunction (OR), but this effect is only observed in the "comprehension" level. The interaction between negation and disjunction not only sheds light on a factor that contributes to code comprehension difficulty, but also reveals why negation itself can be particularly challenging. This experiment demonstrates that there are meaningful differences between distinct levels of code comprehension, and more importantly, that certain cognitive factors may influence only a specific level. These findings underscore the importance of formally defining the level of understanding that a code comprehension experiment aims to evaluate, and caution against generalizing findings from one level of comprehension to another.

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. But there are different levels of understanding. We define and empirically compare two distinct levels: 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. We apply this distinction to the understanding of Boolean expressions, which control the flow of the execution. Our goal is to examine the effect of various factors on understanding such expressions, leading to guidelines for selecting the most understandable version from among 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. 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 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

Aviad Baron
Department of Computer Science
The Hebrew University of Jerusalem, Jerusalem 91904, Israel
Tel.: +972-2-549-4555
E-mail: aviad.baron@mail.huji.ac.il

Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem, Jerusalem 91904, Israel

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 [35, 49]. 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 [34]. 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 [22, 16, 23, 30, 37, 45, 46, 32, 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 [20] 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 to *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 *deriving the general behavior* of the code, beyond a 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, nor any that provide 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. 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 circuit, where they are used for the expression of any given circuit using only NAND gates [47]. 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 shortcut when reading expressions during code tracing, revealing some use of shortcuts 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.

2 Related Work

Programming is hard and intellectually challenging [18,39]. Numerous studies have looked into what makes it hard, especially from the perspective of teaching programming to novices [8,41,44,42]. Many have focused on specific programming constructs, such as loops and recursion [29,43,31,6,25,3]. Suggestions for code complexity metrics have been based on counting such constructs [34,10]. In particular, this has included the logical operators used to create compound logical expressions [34,14].

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 [19]. 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 [26]. 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 [24]. 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 [29]. The study by Herman et al. cited above also included an investigation of students' difficulties in writing Boolean expressions [26]. 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 using 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. [12,36,33]. 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 [35,49]. 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 and processed by programmers [21,17,9,11]. A significant body of research has been performed on the factors influencing the difficulty of understanding code in the past 40 years [7,48].

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. [2,5,38]. As noted above, other definitions of understanding were suggested by Feitelson [20], 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 [28]. 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).

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 between 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 between different expressions that are actually 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 perform "shortcuts"?** Shortcuts refer to the possibility of determining the outcome without considering the full expression. For example,

Table 1 The 16 basic expressions used in the experiment.

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

in an expression with two literals[1] connected by the logical AND operator, if the first literal evaluates to FALSE, there is no need to check the truth 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 [40]. 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. 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

[1] To clarify our terminology: 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.

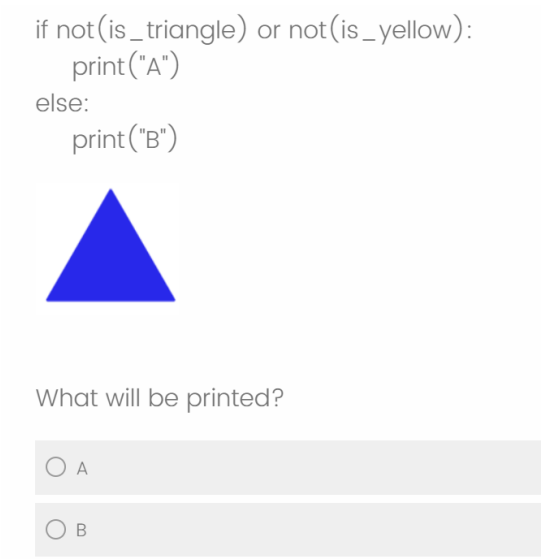


Fig. 1 Example question from part 1 of the experiment.

included 16 questions in total, with each participant receiving 8 of them. Each 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 the table). 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 signif-

icant 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 be claimed to be another confounding factor that influences the process of understanding.

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 deep understanding as the developer's ability to grasp what the expression would return for *all* possible inputs. In other words, it refers to the 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 specific 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,

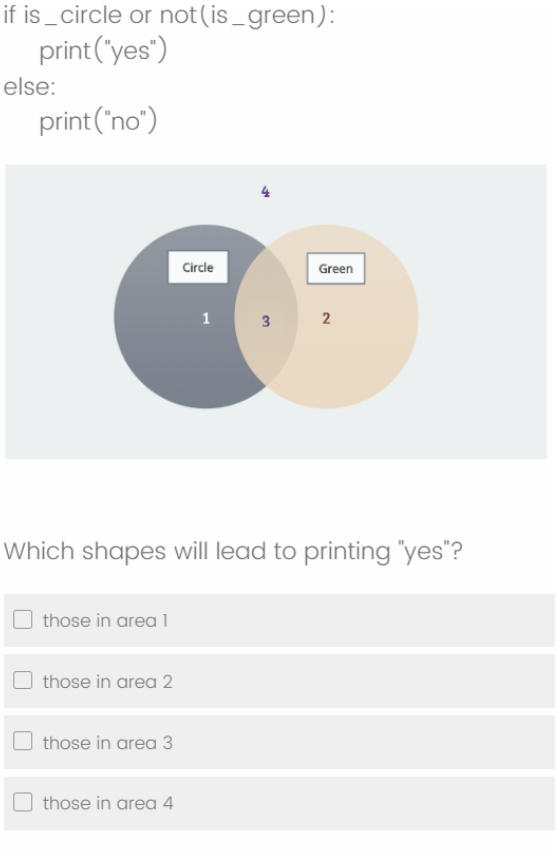


Fig. 2 Example question from part 2 of the experiment.

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 deep 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. However, we believe that in the balance this is a good way to test a more abstract level of understanding.

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.

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.

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. 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, 25% had a Master's degree, and 6% had 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[2].

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

Table 2 Summary of results for all expressions. Comparisons 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%	0.8059	21.61	55%	0.0013*
$\neg(p \wedge q)$	8.38	88%		13.68	68%	
$\neg p \wedge \neg q$	9.25	96%	0.0054	15.76	95%	0.1608
$\neg(p \vee q)$	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%	0.0348
$\neg(\neg p \wedge q)$	11.85	85%		22.89	51%	
$\neg p \vee q$	7.19	93%	$< 10^{-15}$*	20.94	52%	0.8387
$\neg(p \wedge \neg q)$	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%	0.2079	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

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 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.

In addition, Table 2 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.

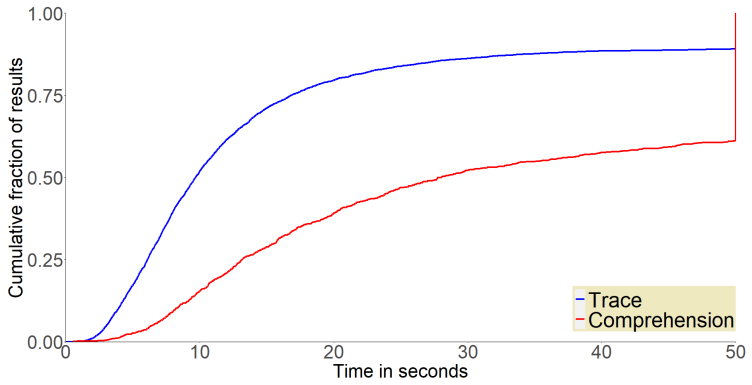


Fig. 3 CDFs of the time to correct answers for all logical expressions for the trace questions (part 1) compared to the comprehension questions (part 2).

5.1 Level of Comprehension

First, we compare between the response times in the tracing part and the times in the comprehension part, which is the subject of RQ1. In Figure 3 and the bottom of Table 2, you can observe that the results are highly significant: tracing expressions is substantially faster and leads to much fewer errors than comprehension of the expressions. This considerable gap clearly indicates a fundamental difference between the types of reading and understanding. Tracing appears to demand less cognitive effort and is significantly easier.

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 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.

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 2 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

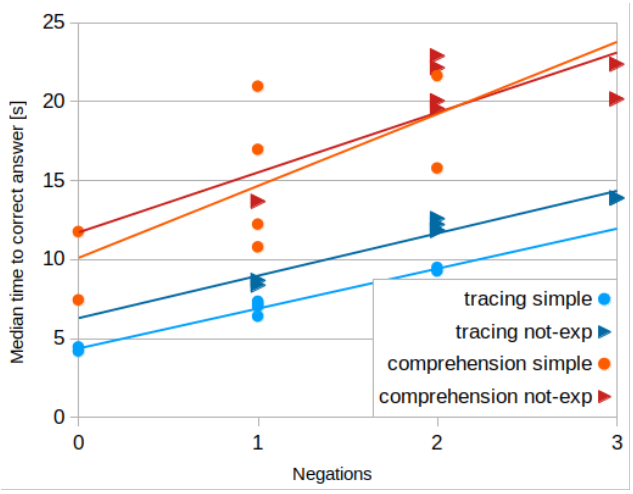


Fig. 4 Linear models of time to correct answer as a function of the number of negations and the expression structure. Simple: a simple condition of two literals. Neg-exp: negation applied to a simple expression.

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 sematic 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.

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.

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.

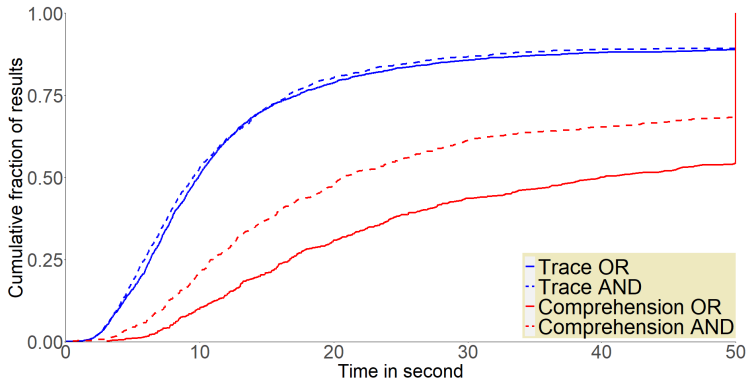


Fig. 5 CDFs of the time to correct answers for AND and OR operators in both tracing and comprehension levels.

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 (p-value = 0.2079), there is a notable gap between them at the comprehension level, with the AND operator being significantly easier to comprehend (both faster and fewer errors, p-value = 1.044e-11). This suggests that while OR is on average not 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.

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 2. 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 = 0.8059), whereas for the second pair there is a significant albeit slight difference (p-value = 0.005399). However, at the comprehension level, there are differences in both pairs—but

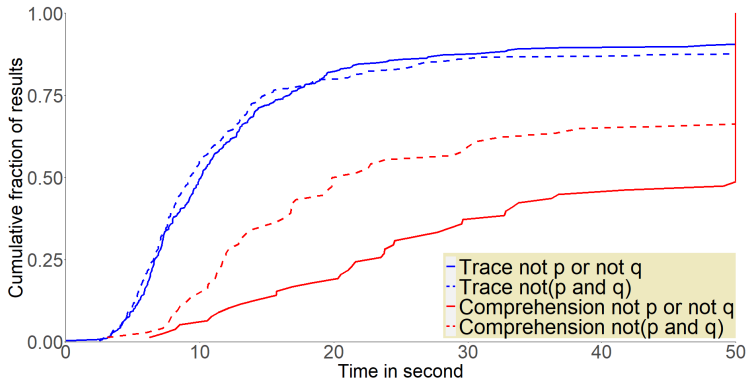


Fig. 6 CDFs of the time to correct answers for the equivalent logical expressions $\neg(p \wedge q)$ and $\neg p \vee \neg q$, in both tracing and comprehension levels.

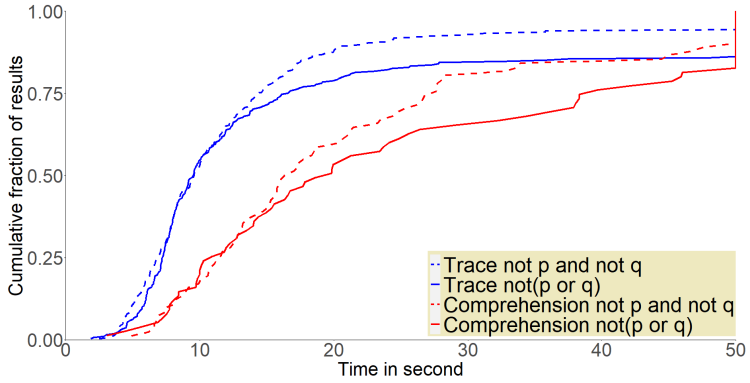


Fig. 7 CDFs of the time to correct answers for the equivalent logical expressions $\neg(p \vee q)$ and $\neg p \wedge \neg q$, in both tracing and comprehension levels.

for the second pair, it is only for part of the distribution and therefore not statistically significant (p-value = 0.001295, p-value = 0.1608). 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 include more negations. We will present only two of these, as the other

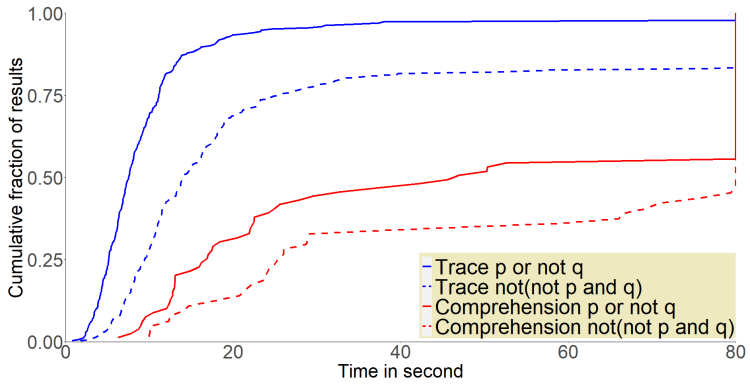


Fig. 8 CDFs of the time to correct answers for the equivalent logical expressions $p \vee \neg q$ and $\neg(\neg p \wedge q)$, in both tracing and comprehension levels.

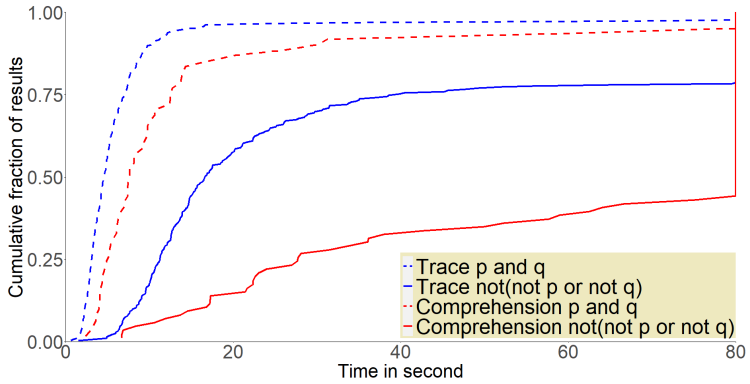


Fig. 9 CDFs of the time to correct answers for the equivalent logical expressions $p \wedge q$ and $\neg(\neg p \vee \neg q)$, in both tracing and comprehension levels.

patterns are similar. In Figure 8, the result for expressions $p \vee \neg q$ and $\neg(\neg p \wedge q)$ 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 $=$ 0.03484). 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 also in the tracing level, and 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

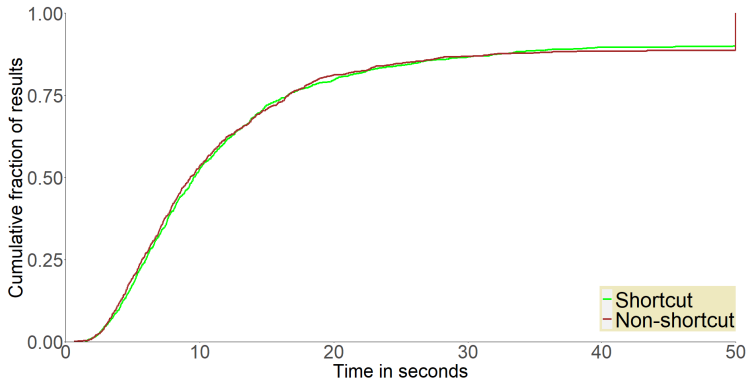


Fig. 10 CDFs of the time for all logical expressions with AND with possibility of shortcuts compared to those without this option.

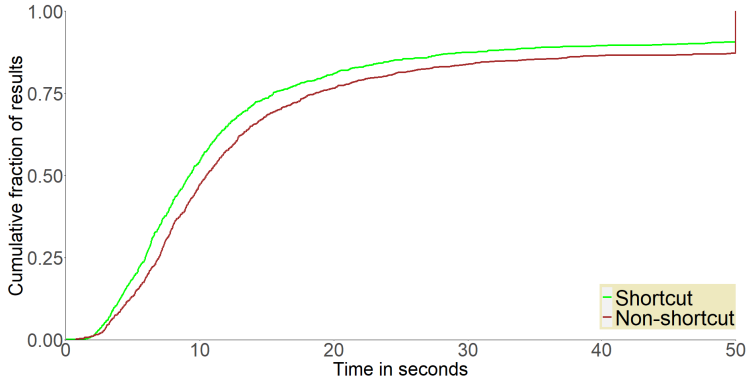


Fig. 11 CDFs of the time for all logical expressions with OR with possibility of shortcuts compared to those without this option.

- 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 Shortcuts

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 whether shortcuts could be used or not, and how significant this was.

Interestingly, we found that using shortcuts also depends on the logical operator (Figures 10 and 11). When the operator is AND, there is absolutely no difference (p-value = 0.2079) between the tracing of expressions that allow shortcuts and those that do not. But for expressions with OR there is a small but statistically significant difference (p-value = 0.000549) between expressions that allowed shortcuts and those that did not. Our conclusion is that there is a tendency to use shortcuts, but only for conditions with OR, and that with only two literals, the effect size is not very substantial.

The reason OR operators are more frequently subject to shortcuts may lie in the fact that a shortcut with OR occurs when the first literal evaluates to TRUE, thereby making the entire expression TRUE. In contrast, for shortcuts involving the AND operator, the shortcut is applied when the 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 shortcuts 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 shortcuts more pronounced. Additionally, in this study, 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 shortcut 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 (e.g. Figure 13 below). In an attempt to understand these results, we analyzed the common mistakes observed for different expressions. This allows 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.

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

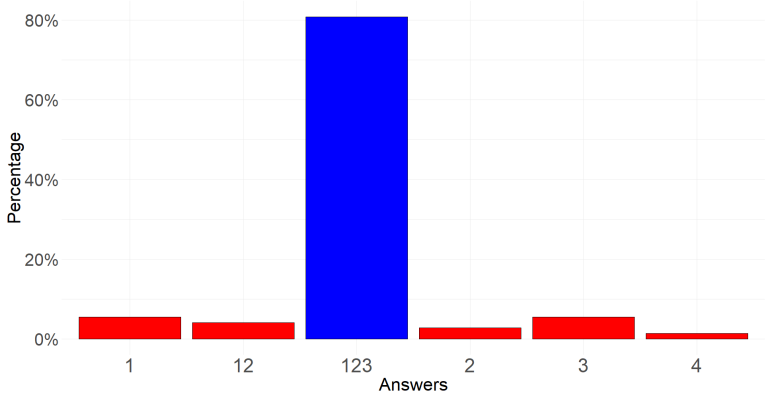


Fig. 12 Responses to comprehension of $p \vee q$. Numbers refer to the areas in the Venn diagram of Figure 2. The correct answer is in blue, and mistakes in red. In this case, no specific mistakes stand out.

the logical XOR operator [26,24]. 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 [19]. 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.

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 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.

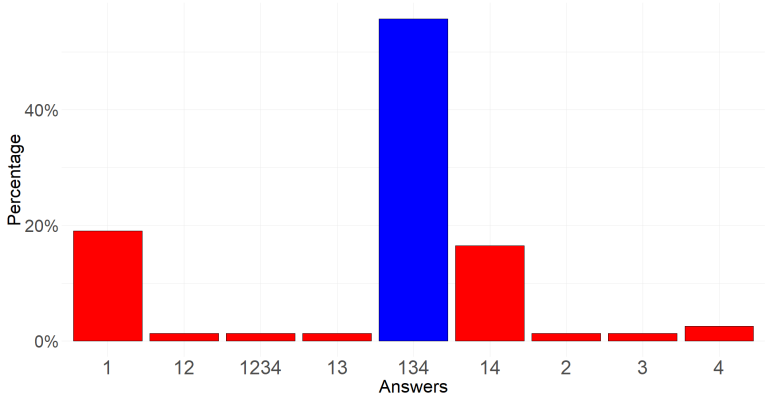


Fig. 13 Responses to comprehension of $p \vee \neg q$.

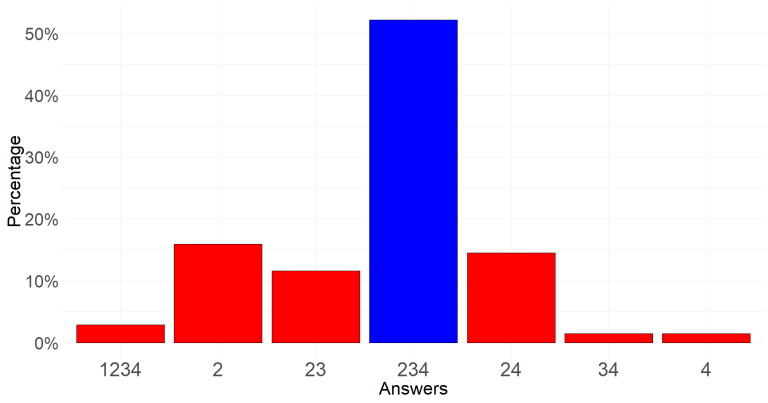


Fig. 14 Responses to comprehension of $\neg p \vee q$.

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.

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.

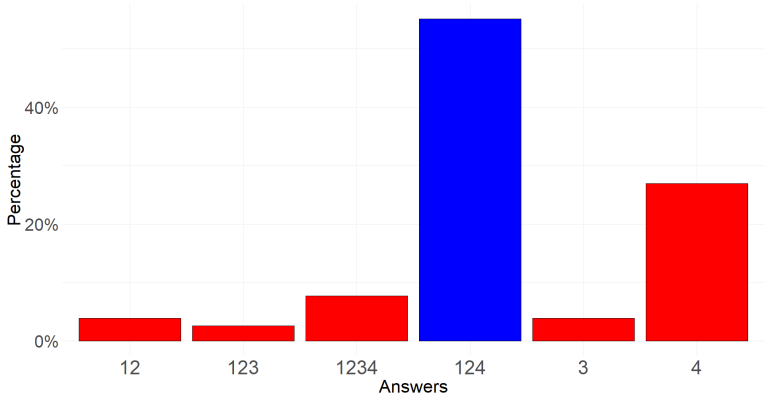


Fig. 15 Responses to comprehension of $\neg p \vee \neg q$.

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 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

Table 3 Common mistakes and their possible explanations. Answers and mistakes are noted by their areas in the Venn diagram of Figure 2. Common mistakes are those that occurred in at least 8% of the answers. "–" means the explanation is inapplicable in this case.

			Explained by						
Expression	correct answer	common mistakes	OR as XOR	mix OR AND	contradict NOT	bad De Morgan	forget NOT()	forget area 4	remain 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	–	

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 3 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.

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 common error observed is identical to the common error observed for the internal expression, and apparently the external NOT was overlooked.
- The common error observed is the complement of the common error that occurs in the internal expression, indicating that the error was made and then the NOT was applied.
- No error was made in the internal expression, but there was a failure to apply the external NOT.

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 [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. 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 shortcuts,

which can be applied to logical conditions. As we have seen, there is some tendency to make these shortcuts. Therefore, experiments based on tracing should account for the possibility of shortcuts 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 canvas 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 [40], 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 question 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 expression 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 [15,27]. 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.

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 2014, for one year. The approval was extended for another year on 8 January 2025.

10.3 Informed consent

We acquired inform consent by presenting the following text in the begging 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 Softw. Eng.*, 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 and Technology in Computer Science Education*, 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. 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.
15. Viviane Déprez and M. Teresa Espinal, editors. *The Oxford Handbook of Negation*. Oxford University Press, 2020.
16. 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.

17. E. W. Dijkstra. Go To statement considered harmful. *Comm. ACM*, 11(3):147–148, Mar 1968.
18. Edsger W. Dijkstra. The humble programmer. *Comm. ACM*, 15(10):859–866, Oct 1972.
19. Alireza Ebrahimi. Novice programmer errors: Language constructs and plan composition. *Intl. J. Human-Computer Studies*, 41(4):457–480, Oct 1994.
20. Dror G. Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Softw. Eng.*, 27(6), Nov 2022.
21. 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.
22. Yosef Grodzinsky et al. Logical negation mapped onto the brain. *Brain Structure and Function*, 35:19–31, 2020.
23. Yosef Grodzinsky et al. A linguistic complexity pattern that defies aging: The processing of multiple negations. *Journal of Neurolinguistics*, 58:543–554, 2021.
24. 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.
25. 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.
26. 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.
27. Laurence R. Horn, editor. *The Expression of Negation*. De Gruyter Mouton, 2010.
28. 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.
29. Errol R. Iselin. Conditional statements, looping constructs, and program comprehension: An experimental study. *Intl. J. Man-Machine Studies*, 28(1):45–66, Jan 1988.
30. Marcel Adam Just and Patricia Ann Carpenter. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior*, 10:244–253, 1971.
31. Claudius M. Kessler and John R. Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Comput. Interaction*, 2(2):135–166, 1986.
32. Sangeet Khemlani, Isabel Orenes, and P.N. Johnson-Laird. The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica*, 151:1–7, 2014.
33. 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.
34. Thomas J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308–320, Dec 1976.
35. Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Intl. Conf. Program Comprehension*, number 23, pages 25–35, May 2015.
36. Francisco Moretti. Avoid negative conditionals. URL https://www.franciscomoretti.com/blog/avoid-negative-conditionals, 5 Jun 2023.
37. 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.
38. 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.
39. Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Edu.*, 18(1), Oct 2017.
40. Václav Rajlich and George S. Cowan. Towards standard for experiments in program comprehension. In *IEEE Intl. Workshop Program Comprehension*, number 5, pages 160–161, Mar 1997.

41. Tamarisk Lurlyn Scholtz and Ian Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Conf. Innovation & Tech. in Comput. Sci. Educ.*, number 15, pages 103–107, Jun 2010.
42. 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.
43. Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Comm. ACM*, 26(11):853–860, Nov 1983.
44. Juha Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Edu.*, 13(2), Jun 2013.
45. Ye Tian and Richard Breheny. Dynamic pragmatic view of negation processing. *Negation and Polarity: Experimental Perspectives*, 1:21–43, 2015.
46. P. C. Wason. The processing of positive and negative information. *Quarterly Journal of Experimental Psychology*, 11:92–107, 1959.
47. Niklaus Wirth. *Digital Circuit Design for Computer Science Students: An Introductory Textbook*. Springer-Verlag, 1995.
48. 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.
49. 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.

2.4 Additional papers

During the course of the doctoral research, two additional papers were written:

- Aviad Baron and Dror G. Feitelson. How A Data Structure's Linearity Affects Programming and Code Comprehension: The Case of Recursion vs. Iteration. PPIG 2023, Lund, Sweden.

- Aviad Baron and Dror G. Feitelson. Why Is Recursion Hard to Comprehend? An Experiment with Experienced Programmers in Python. ITiCSE 2024: Milan, Italy.

However, they are not included in the dissertation, as they address different topics and not negation.

Chapter 3

Discussion and Conclusions

3.1 Contributions

In this study, we set out to map the impact of negation on the comprehension of Boolean expressions in code. Our work offers contributions on several levels. First, we extend the research on negation in the context of natural language into a novel domain—that of source code. The world of code combines elements of natural language (e.g., in variable names) with formal mathematical logic. This combination creates a unique context in which rich cognitive interactions can emerge. For example, due to the formal structure of code, certain cognitive factors such as regularity are more easily identifiable than in natural language, thereby allowing us to contribute new cognitive insights regarding the role of negation. Furthermore, the interaction between different types of negation is a phenomenon that appears to be unique to code and does not typically arise in natural language.

A second level of contribution lies in the domain of program comprehension. Boolean expressions are a central component of the logic and functionality of code. Therefore, the way in which logical operators are written, how Boolean variables are named, and how expressions are structured can critically affect code readability. Developers spend a significant portion of their time reading and understanding existing code—a trend that is only intensifying with the growing use of generative AI tools. It is highly important that code be as clear as possible, enabling a deep understanding of its behavior to ensure functional correctness.

One may claim that actually writing better code is becoming less important, as most code is—or soon will be—AI-generated. But this stand ignores the fact that AI tools are based on harvesting huge amounts of existing code. Refactoring generated code and improving it is therefore a crucial feedback-loop for improving the quality of the next generation of such tools. Recommendations concerning how to make code more readable are therefore still important, even when most code is not written from scratch by humans.

Accordingly, another major contribution of our work is the ability to evaluate higher-order comprehension of code and to distinguish between different levels of understanding, with a particular emphasis on deeper semantic comprehension. These findings have practical implications for code refactoring and for writing new or revised code in a clearer, more maintainable form.

3.2 Conclusions

This dissertation investigates how developers comprehend Boolean expressions in code, with a particular focus on the role of negation—a construct known to increase cognitive complexity both in natural language and formal logic. Through a series of empirical studies, we examine the cognitive processes underlying this form of understanding, map out key syntactic and logical factors that influence comprehension, and draw distinctions between different levels of understanding. Collectively, the work contributes to foundational knowledge in program comprehension and provides practical recommendations for writing more readable and maintainable code.

We begin by extending insights from the domain of natural language negation to the domain of programming languages. Unlike natural language, code embodies a hybrid structure, combining formal logical constructs (e.g., !, !=, AND, OR) with natural language elements (e.g., variable names such as notDone or isEmpty). This fusion creates a novel and highly structured context that not only permits the identification of new cognitive factors—such as syntactic and logical regularities—but also reveals complex interactions that are less accessible in natural language contexts.

One of the central findings of this dissertation is that negation—regardless of its form—consistently increases the difficulty of code comprehension, both in terms of time and error rates. These effects are magnified when multiple forms of negation interact, such as when negated variable names are combined with explicit negation operators. These insights are supported by concrete experimental evidence, showing that even simple transformations (e.g., rewriting !(length == 0) as length > 0) significantly improve understanding. Furthermore, expressions involving double negation or lack of structural regularity result in disproportionate cognitive load, highlighting the importance of code clarity and consistency.

Beyond individual factors, our research also distinguishes between two levels of comprehension:

- Tracing, which refers to evaluating a condition for a specific input; and
- Comprehension, which involves generalizing the behavior of the condition across all possible inputs.

Our experiments reveal that these levels are not interchangeable: conditions that appear equally simple at the tracing level can present drastically different levels of difficulty at the comprehension level. For instance, while no significant difference was observed between AND and OR expressions in tracing tasks, some OR expressions with negations were significantly harder to understand in comprehension tasks. These findings emphasize the necessity for clearly defining the cognitive level under examination in program comprehension studies, and suggest that current research may over-rely on surface-level tracing tasks that fail to capture the deeper complexities of semantic understanding.

This work also opens multiple avenues for future research. Potential directions include examining comprehension of mixed AND/OR expressions, exploring the effect of programming experience, analyzing conditions involving numerical comparisons (e.g., x <= 3 vs. !(x > 3)), and investigating how short-circuit evaluation influences cognitive processing. There is also room for additional work on how negations in variable names affect comprehension, and how developers reason about these during code reviews or refactoring. In addition, this study was the first to compare between different levels of code comprehension. Future work may extend this line of research by refining the definitions of these comprehension levels, and by designing experiments that explore their cognitive implications across diverse contexts, beyond the specific domain of Boolean expressions.

Finally, this dissertation has practical implications for coding practices and pedagogy. It offers actionable guidelines for writing clearer Boolean expressions: avoiding unnecessary negations, minimizing double negations, emphasizing regularity, and choosing semantically transparent variable names. These insights are especially relevant in an era where developers frequently interact with AI-generated code, and where maintaining a high level of code readability and correctness is essential.

In sum, this dissertation not only deepens our theoretical understanding of logical condition processing in code, but also bridges the gap between cognitive science and software engineering, suggesting new ways to improve both developer tooling and program comprehension.

Bibliography

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

[2] S. Ajami, Y. Woodbridge, and D. G. Feitelson. Syntax, predicates, idioms — what really affects code complexity? *Empirical Software engineering*, 24(1):287–328, Feb 2019.

[3] A. Baron and D. G. Feitelson. Why is recursion hard to comprehend? an experiment with experienced programmers in Python. In *Innovation and Technology in Computer Science Education*, volume 1, pages 115–121, Jul 2024.

[4] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel. Indentation: Simply a matter of style or support for program comprehension? In *Intl. Conf. Program Comprehension*, number 27, pages 154–164, May 2019.

[5] A. C. Benander, B. A. Benander, and H. Pu. Recursion vs. iteration: An empirical study of comprehension. *J. Syst. & Softw.*, 32(1):73–82, Jan 1996.

[6] R. E. Brooks. Towards a theory of the comprehension of computer programs. *Int. J. Man Mach. Stud.*, 18(6):543–554, 1983.

[7] L. Budde, B. Heinemann, and C. 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.

[8] R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Trans. Softw. Eng.*, 36(4):546–558, Jul/Aug 2010.

[9] G. Cesium. Why to stop writing negative code. URL https://medium.com/@Cuadraman/why-to-stop-writting-negavite-code-af5ffb17195, 23 Nov 2018.

[10] 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.

[11] B. Curtis, J. Sappidi, and J. Subramanyam. An evaluation of the internal quality of business applications: does size matter? In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering,*, pages 711–715. ACM, 2011.

[12] 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.

[13] V. Déprez and M. T. Espinal, editors. *The Oxford Handbook of Negation.* Oxford University Press, 2020.

[14] 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.

[15] E. W. Dijkstra. The humble programmer. *Comm. ACM*, 15(10):859–866, Oct 1972.

[16] A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457–480, Oct 1994.

[17] D. G. Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software engineering*, 27(6), Nov 2022.

[18] Y. Grodzinsky et al. Logical negation mapped onto the brain. *Brain Structure and Function*, 35:19–31, 2020.

[19] Y. Grodzinsky et al. A linguistic complexity pattern that defies aging: The processing of multiple negations. *Journal of Neurolinguistics*, 58:543–554, 2021.

[20] S. Grover and S. 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.

[21] B. Haberman and H. 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.

[22] S. M. Henry and D. G. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.

[23] G. L. Herman, M. C. Loui, L. Kaczmarczyk, and C. Zilles. Describing the what and why of students' difficulties in Boolean logic. *ACM Trans. Comput. Edu.*, 12(1), Mar 2012.

[24] L. R. Horn, editor. *The Expression of Negation*. De Gruyter Mouton, 2010.

[25] N. Hurtig, J. Hollingsworth, S. Blankenship, E. Kraemer, M. Sitaraman, and J. O. Hallstrom. Network visualization and assessment of student reasoning about conditionals. In *Innotaion & Tech. Comput. Sci. Edu.*, number 27, pages 255–261, Jul 2022.

[26] E. R. Iselin. Conditional statements, looping constructs, and program comprehension: An experimental study. *Intl. J. Man-Machine Studies*, 28(1):45–66, Jan 1988.

[27] M. A. Just and P. A. Carpenter. Comprehension of negation with quantification. *Journal of Verbal Learning and Verbal Behavior*, 10:244–253, 1971.

[28] C. M. Kessler and J. R. Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Comput. Interaction*, 2(2):135–166, 1986.

[29] S. Khemlani, I. Orenes, and P. Johnson-Laird. The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica*, 151:1–7, 2014.

[30] D. Lindner. Don't ever not avoid negative logic. URL https://schneide.blog/2014/08/03/dont-ever-not-avoid-negative-logic/, 3 Aug 2014.

[31] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[32] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308–320, Dec 1976.

[33] F. Moretti. Avoid negative conditionals. URL https://www.franciscomoretti.com/blog/avoid-negative-conditionals, 5 Jun 2023.

[34] G. J. Myers. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notices*, 12(10):61–64, 1977.

[35] I. Orenes, L. Moxey, C. Scheepers, and C. Santamaría. Negation in context: Evidence from the visual world paradigm. *Quarterly Journal of Experimental Psychology*, 69, 2016.

[36] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *Intl. Conf. Softw. Eng.*, number 43, pages 524–536, May 2021.

[37] Y. Qian and J. Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Edu.*, 18(1), Oct 2017.

[38] T. L. Scholtz and I. Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Conf. Innovation & Tech. in Comput. Sci. Educ.*, number 15, pages 103–107, Jun 2010.

[39] D. Sleeman, R. T. Putnam, J. Baxter, and L. Kuspa. Pascal and high school students: A study of errors. *J. Edu. Comput. Res.*, 2(1):5–23, Feb 1986.

[40] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Comm. ACM*, 26(11):853–860, Nov 1983.

[41] J. Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Edu.*, 13(2), Jun 2013.

[42] M.-A. D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proc. 13th International Workshop on Program Comprehension*, pages 181–191. IEEE Computer Society, 2005.

[43] Y. Tian and R. Breheny. Dynamic pragmatic view of negation processing. *Negation and Polarity: Experimental Perspectives*, 1:21–43, 2015.

[44] P. C. Wason. The processing of positive and negative information. *Quarterly Journal of Experimental Psychology*, 11:92–107, 1959.

תקציר

שלילה היא מרכיב יסודי בשפה האנושית. כפי שכתב הבלשן Larry Horn: "במובנים רבים, השלילה היא זו שהופכת אותנו לבני אדם, ומעניקה לנו את היכולת להכחיש, לסתור, לעוות, לשקר ולהעביר אירוניה". עיבוד של משפטים עם וללא שלילה, לרבות מופעים של שלילה כפולה ושל יחסים לוגיים מגוונים, העסיקו לא מעט בלשנים, אנשי מדעי המוח ופילוסופים. במחקרים במדעי המוח אף נעזרו בכלים כגון הדמיית fMRI למיפוי תהליכי העיבוד הלוגי לאזורים ייעודיים במוח, ובכך סיפקו תובנות על האופן שבו המוח האנושי מנווט ומבין מבנים לוגיים מורכבים.

עבודת דוקטורט זו בוחנת את האתגרים הקוגניטיביים הכרוכים בהבנת ביטויים בוליאניים עם שלילות בקונטקסט של קוד, תוך התמקדות בשיפור הקריאות וההבנה של קוד. המחקר כולל שלושה מחקרים משולבים, שכל אחד מהם עוסק בהיבט ייחודי של לוגיקה בוליאנית ובהשפעתו על הבנת תוכניות, וביחד הם מספקים הבנה מעמיקה יותר של הגורמים המשפיעים על עיבוד קוגניטיבי בעת ניתוח קוד.

המחקר הראשון בוחן כיצד גורמים ספציפיים כגון שלילות, רגולריות(עקביות) וערכי אמת משפיעים על הבנת ביטויים לוגיים. בניסוי שנערך עם 205 מפתחים מקצועיים נמצא כי ביטויים הכוללים שלילות רבות דורשים זמן עיבוד ארוך יותר, וכמו כן שלילה כפולה מתבררת כמאתגרת במיוחד. עם זאת, זוהו גם גורמים נוספים המשפיעים על ההבנה. לדוגמה, ליטרלים שערכם אמת עובדו מהר יותר מליטרלים שערך האמת שלהם שקר. כמו כן, רגולריות, מצב שבו כל המשתנים מופיעים עם שלילה או שכולם מופיעים בלעדיה, או מצב שבו כל הליטרלים הם בעלי ערך אמת או כולם בעלי ערך שקר, נמצאה כגורם משמעותי בהבנה. עם זאת, קיימות אינטראקציות מורכבות בין הגורמים, המובילות לתוצאות מורכבות יותר.

המחקר השני מתמקד במגוון סוגי השלילה בקוד. בפרט, נבחנים סוגים שונים של אופרטורים בשפות תכנות המבטאים שלילה, וכן שמות משתנים הכוללים שלילה. כדי לבדוק האם השימוש בסוגים שונים של ביטויי שלילה משפיע על הבנת קוד, נערך ניסוי מבוקר בהשתתפות 268 נבדקים, שנדרשו להבין קטעי קוד קצרים המכילים ביטויים לוגיים שונים וצורות של שלילה. הממצאים הראו הבדלים מובהקים בהבנה: הן בדיוק והן בזמן הביצוע כתלות בסוג ובשילוב של השלילות. שילובים מורכבים של סוגי שלילה שונים הגבירו את העומס הקוגניטיבי והפחיתו את שיעור התשובות הנכונות. ממצאים אלו מדגישים את חשיבות הפשטת דפוסי השלילה בקוד לצורך שיפור הקריאות, ומספקים תובנות יישומיות להנחיות לכתיבת קוד קריא.

המחקר השלישי עוסק בהבחנה רחבה יותר בהבנת תוכניות, תוך הגדרה והשוואה בין שני סוגי הבנה של ביטויים בוליאניים: **מעקב**, כלומר, היכול לעקוב צעד־אחר־צעד אחר הרצת התוכנית ו**הבנה מהותית**, דהיינו, היכולת לתפוס את משמעות הקוד וההתנהגות הכללית שלו. בניסוי מבוקר בהשתתפות 362 נבדקים נמצא כי הבנה מהותית דרשה זמן ארוך יותר והייתה מועדת יותר לטעויות בהשוואה למעקב ביצוע. עוד נמצא כי ביטויים הכוללים את האופרטור הלוגי "או" היו קשים יותר להבנה לעומת כאלה הכוללים" וגם", אך הבדל זה הופיע רק ברמת ההבנה המהותית. בנוסף, נמצאו הבדלים בהבנה בין ביטויים שקולים לוגית, כאשר חלק מהצורות הסינטקטיות היו אינטואיטיביות יותר מאחרות. ניתוח מעמיק של

הטעויות הנפוצות מראה שהן אינן אקראיות: קיימת אינטראקציה שיטתית בין האופרטור "או", לבין שלילה, ושילובם יוצר מורכבות לוגית משמעותית.

בסיכומם של דברים, שלושת המחקרים מציעים תמונה קוהרנטית ורבת מימדים של האתגרים הקוגניטיביים הכרוכים בהבנת ביטויים בוליאניים. הממצאים מדגישים את השפעתם המשמעותית של גורמים מבניים, סינטקטיים והקשריים, כגון שלילה, סדירות וסוג האופרטור, על יכולתם של מפתחים להבין קוד בצורה יעילה ומדויקת. עבודה זו תורמת לקידום ההבנה המדעית של קריאות קוד, ומציעה המלצות מעשיות לכתיבה בהירה ותחזוקה נוחה יותר של קוד. תוך שילוב ממדים יישומיים ותיאורטיים, היא מרחיבה את הידע האקדמי על עיבוד קוגניטיבי של שלילה בהקשר הייחודי והחדשני של קוד, ומצביעה על אפשרויות ליישום הממצאים בפרקטיקות פיתוח מודרניות, לרבות כלים ל-refactoring וניתוח אוטומטי של קריאות קוד.

עבודה זו נעשתה בהדרכתו של פרופ' דרור פייטלסון

לא קל להבנה: על שלילות בקוד והשפעתן על הבנת הקוד

חיבור לשם קבלת תואר דוקטור לפילוסופיה

מאת

אביעד ברון

הוגש לסנאט האוניברסיטה העברית בירושלים

8/2025