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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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

© 2024 Copyright held by the owner/author(s).

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:

```
1 is_green = False
2 is_card = False
3 is_circle = True
4 if is_green or is_card or not(is_circle):
5     print("A")
6 else:
7     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:

```
1 is_blue = False
2 is_wet = False
3 is_card = False
4 if not(is_blue) and not(is_wet) and not(is_card):
5     print("A")
6 else:
7     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

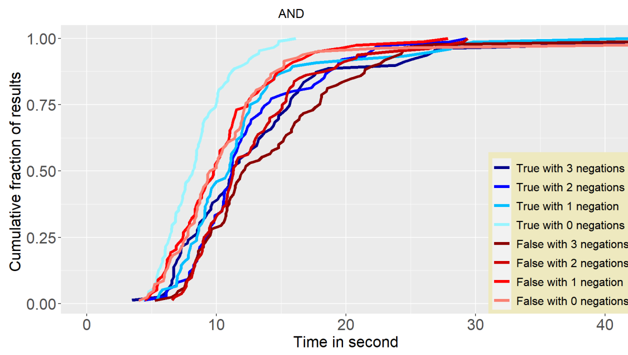

Figure 1: CDFs of the time to correct answers for logical expressions with 3 literals connected by AND operators.

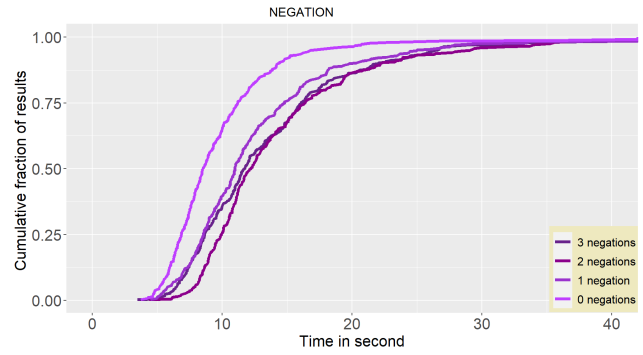

Figure 3: CDFs of the time to correct answers for logical expressions with different numbers of negated variables.

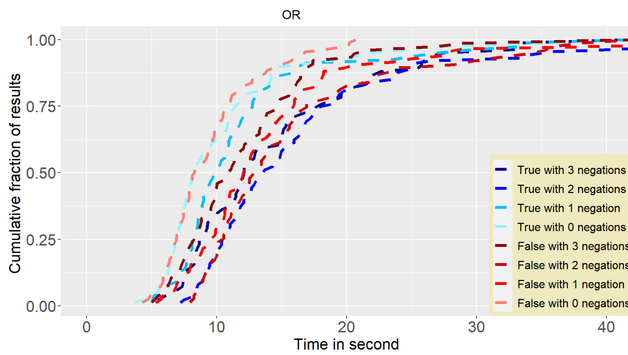

Figure 2: CDFs of the time to correct answers for logical expressions with 3 literals connected by OR operators.

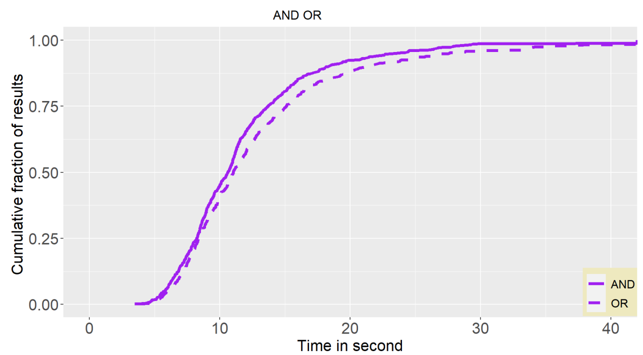

Figure 4: CDFs of the time to correct answers for logical expressions with different operators.

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

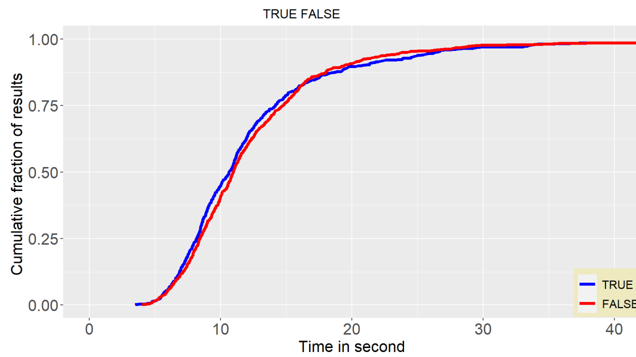

Figure 5: CDFs of the time to correct answers for logical expressions with different truth values.

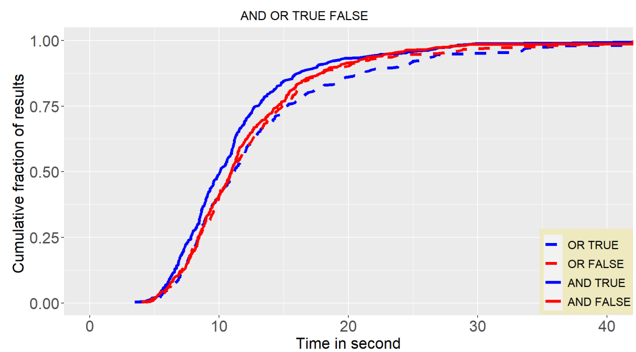

Figure 6: CDFs of the time to correct answers for logical expressions with different operators and truth values.

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

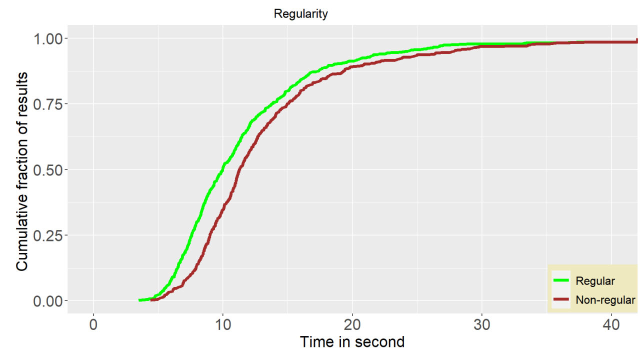

Figure 7: CDFs of the time to correct answers for regular logical expressions vs. irregular logical expressions.

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

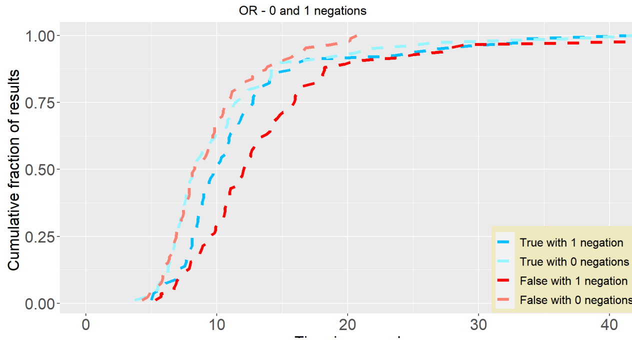

Figure 8: CDFs of the time to correct answers for expressions based on OR with 0 or 1 negations.

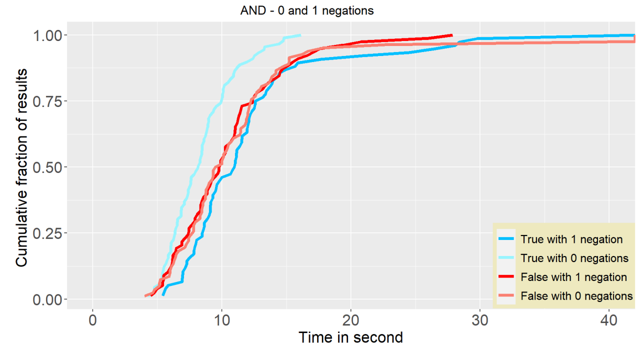

Figure 9: CDFs of the time to correct answers for expressions based on AND with 0 or 1 negations.

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 $\neg(p \land q)$:

```
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 $\neg p \lor \neg q$:

```
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")
```

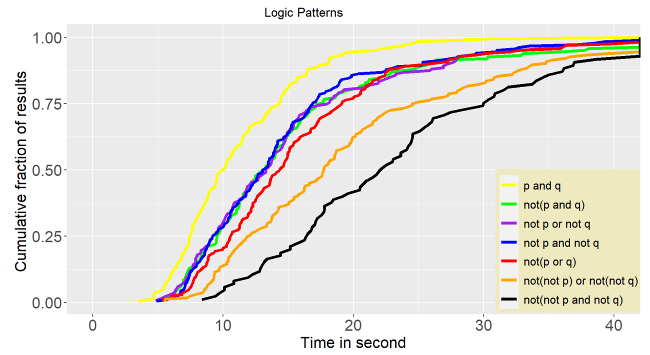

Figure 10: CDFs of the time for comprehending the seven snippets. All had the value TRUE except $p \land q$, which was FALSE and represents an implicit negation.

(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: $\neg(p \land q)$ and the logically equivalent $\neg p \lor \neg q$
- De Morgan's second pair: $\neg(p \lor q)$ and the logically equivalent $\neg p \land \neg q$
- The "implicit negation" when the condition is simply $p \land q$ but it receives a false value.
- Expressions with double negations, represented by the condition $\neg(\neg p \land \neg q)$ and its corresponding expression $\neg\neg p \lor \neg\neg q$.

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 $p \land q$ that evaluated to FALSE. This was significantly faster than the logically equivalent $\neg(p \land q)$ that evaluated to TRUE, but had a negation (t-test p-value = 7.803e-07).

In general, all four expressions: $\neg(p \land q)$, $\neg p \lor \neg q$, $\neg(p \lor q)$, and $\neg p \land \neg q$ are quite similar in terms of their processing difficulty, with a slightly higher difficulty for the condition $\neg(p \lor q)$. Performing the t-test on De Morgan pairs, we find that there are no statistically significant differences. For the pair $\neg(p \land q)$ and $\neg p \lor \neg q$, which look like exactly the same time distribution, the result was p-value = 0.5916. But also for the pair $\neg(p \lor q)$ and $\neg p \land \neg q$, 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