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

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

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

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 3–6, 2025, Charlottesville, VA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3702652.3744213

1 Introduction

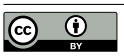
Negation is a fundamental feature of human language. As linguist Larry Horn writes [12], "In many ways, negation is what makes us human, imbuing us with the capacity to deny, to contradict, to misrepresent, to lie, and to convey irony." The processing of sentences with and without negation, including 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 979-8-4007-1340-8/2025/08
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.

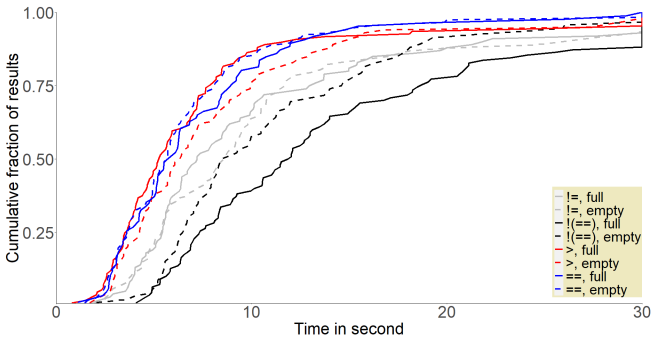


Figure 1: CDFs of the time to correct answers for logical expressions of part 1.

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.

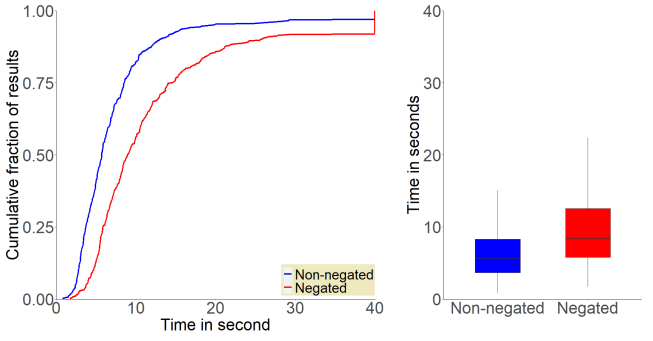


Figure 2: CDFs and boxplots of time to correct answers for negated vs. non-negated logical expressions. The CDFs include incorrect answers as ∞, the boxplots do not.

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.

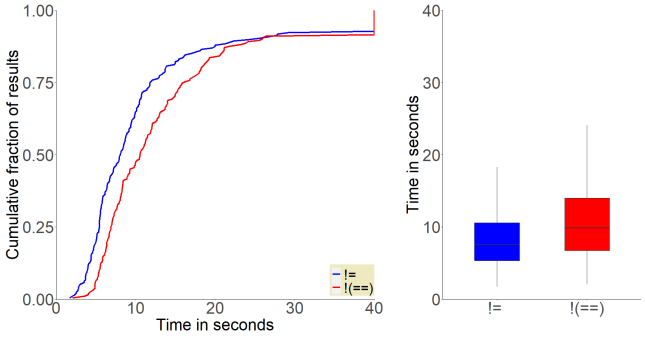


Figure 3: CDFs and boxplots of of the time to correct answers for logical expressions with != vs. !(==).

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.

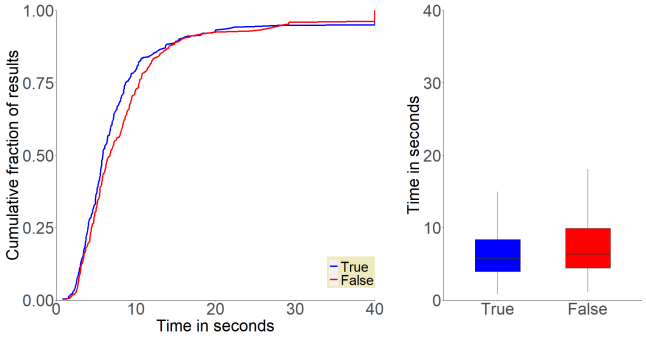


Figure 4: CDFs and boxplots of the time to correct answers for logical expressions with different truth values.

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`.

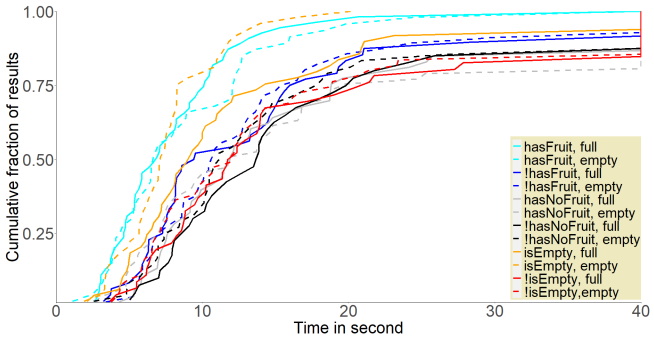


Figure 5: CDFs of the time for comprehending the 12 snippets in part 2.

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`.

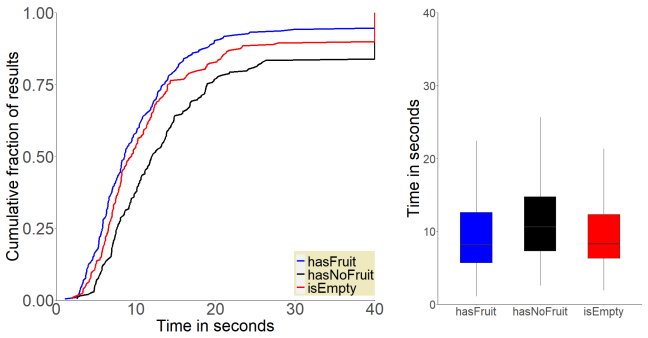


Figure 6: CDFs and boxplots of the time to correct answers for logical expressions with different variables names.

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.

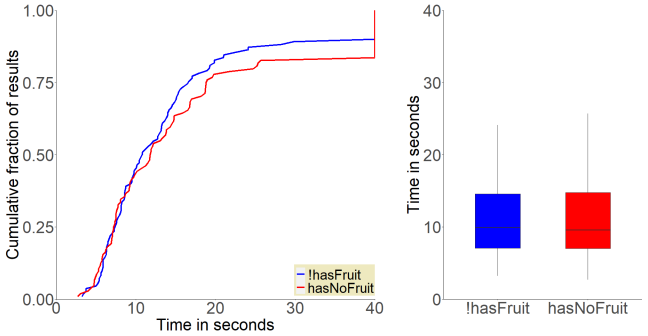


Figure 7: CDFs and boxplots of the time to correct answers for negated naming versus the negation operator.

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.

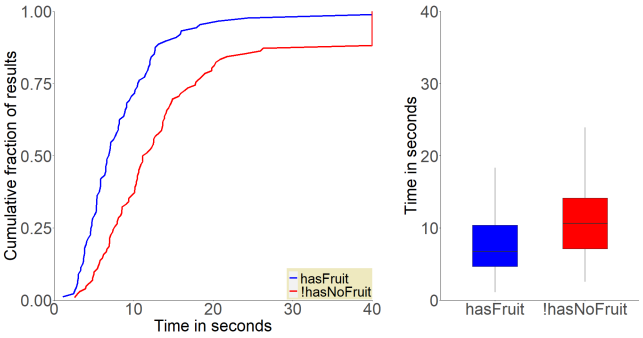


Figure 8: CDFs and boxplots of the time to correct answers for a double negated vs. non-negated expression.

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.

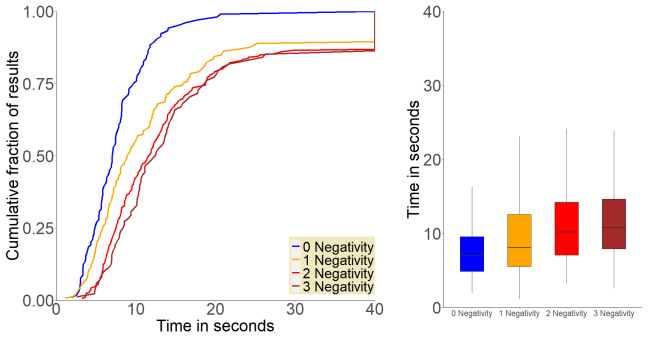


Figure 9: CDFs and boxplots of the time to correct answers for code snippets with different negativity counts.

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

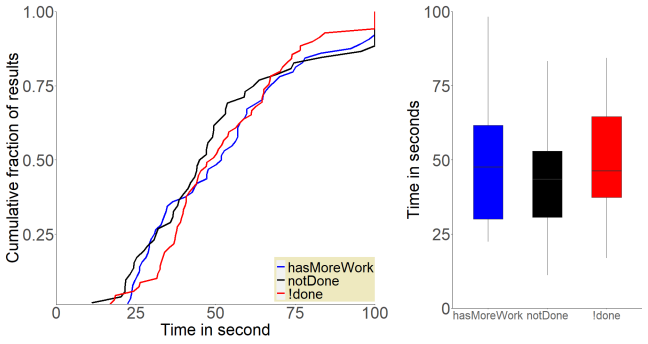


Figure 10: CDFs and boxplots of the time to correct answers for the first set of `while` snippets.

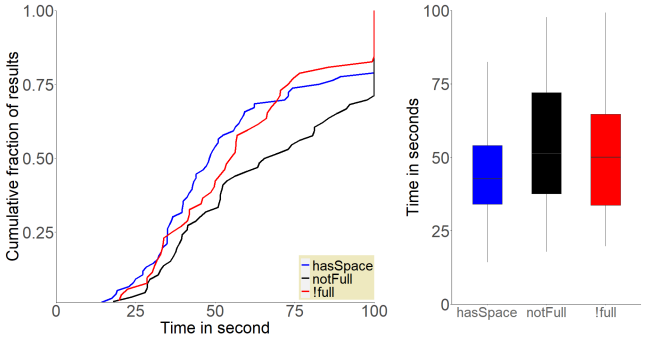


Figure 11: CDFs and boxplots of the time to correct answers for the second set of `while` snippets.

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. 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] Richard A. Armstrong. 2014. When to use the Bonferroni correction. *Ophthalmic & Physiological Optics* 34 (2014), 502–508. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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). https://doi.org/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). https://doi.org/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. https://doi.org/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). https://doi.org/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. https://doi.org/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. https://doi.org/10.1109/TSE.2017.2734091