The Journal of Systems & Software 203 (2023) 111731

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An Investigation of confusing code patterns in JavaScript[☆]

Adriano Torres [a], Caio Oliveira [a], Márcio Okimoto [a], Diego Marcílio [b], Pedro Queiroga [c], Fernando Castor [c,d], Rodrigo Bonifácio [a,*], Edna Dias Canedo [a], Márcio Ribeiro [e], Eduardo Monteiro [f]

[a] *Computer Science Department, University of Brasília, Brazil*
[b] *Universitá Della Svizzera Italiana, Switzerland*
[c] *Informatics Center, University Federal of Pernambuco, Brazil*
[d] *Information and Computing Sciences Department, Utrecht University, The Netherlands*
[e] *Institute of Computing, Federal University of Alagoas, Brazil*
[f] *Statistics Department, University of Brasília, Brazil*

ARTICLE INFO

Article history:
Received 31 July 2022
Received in revised form 17 April 2023
Accepted 25 April 2023
Available online 9 May 2023

Dataset link: https://github.com/rbonifacio/AtomsJS/tree/main/package

Keywords:
Software maintenance
Program comprehension
Program understanding
Atoms of confusion
JavaScript code

ABSTRACT

Evolving software is particularly challenging when the code has been poorly written or uses confusing idioms and language constructs, which might increase maintenance efforts and impose a significant cognitive load on developers. Previous research has investigated possible sources of confusion in programs, including the impact of small code patterns (hereafter atoms of confusion) that contribute to misunderstanding the source code. Although researchers have explored atoms of confusion in code written in C, C++, and Java, different languages have different features, developer communities, and development cultures. This justifies the exploration of other languages to verify whether they also exhibit confusion-inducing patterns. In this paper we investigate the impact of atoms of confusion on understanding JavaScript code—a dynamically typed language whose popularity is growing in the most diverse application domains. We present the results of a mixed-methods research comprising a mining software repositories (MSR) study, two experiments, and a set of interviews with practitioners. Our MSR effort shows that atom candidates are frequent and used intensively in 72 popular open-source JavaScript projects: four atom candidates appear in 90% of them and two of them occur more than once for every 100 lines of code. This helps motivate the other three studies. The results of both experiments suggest that two code patterns that have been previously observed to confuse C programmers also confuse JavaScript programmers: the comma operator and assignments being used as values. In addition, some code patterns, such as omitted curly braces and change of literal encoding, have caused confusion in participants in one of the experiments. We discover that some JavaScript-specific elements, such as automatic semicolon insertion and object destructuring, also have the potential to cause confusion. For all these cases effect sizes were either medium or high. The interviews we conducted indicate other constructs and idioms that merit investigation in the future.

1. Introduction

Developers are often confused while reading unfamiliar code (Ebert et al., 2021). According to Hermans (2021), that confusion can stem from three sources: (i) lack of knowledge, i.e., not knowing what an element in the program does; (ii) lack of information, i.e., not knowing how a program element works; and (iii) lack of processing power, i.e., the inability to combine all the elements in a program and make sense of its execution in one's head. Even small pieces of code can be confusing for developers (Ajami et al., 2017; Gopstein et al., 2017) at first glance.

For instance, consider the pair of JavaScript code snippets in Listing 1 (adapted from NervJS/taro). The code snippet on the left-hand side is arguably harder to understand than the one on the right-hand side. The former uses the logical AND operator (`&&`) beyond its lexical meaning (Castor, 2018) to determine the control flow of the program. Furthermore, the expression has a potential side effect, due to the use of the post-increment operator (`s++`). The snippet on the right-hand side uses an `if` statement for control flow and increments the value of variable `s` using an assignment, making it easier to understand. Notwithstanding, confusion is a direct consequence of the experience of the person who is reading the code. A developer who customarily

[☆] Editor: Christoph Treude.
[*] Corresponding author.
E-mail address: rbonifacio@unb.br (R. Bonifácio).

https://doi.org/10.1016/j.jss.2023.111731
0164-1212/© 2023 Elsevier Inc. All rights reserved.

```
res = ps.reduce((s, b) => {
  b && s++
  return s
}
```

```
res = ps.reduce((s, b) => {
  if(b) {
    s = s + 1
  }
  return s
}
```

Listing 1: Code snippet from NervJS/taro project (left-hand side) and an alternative implementation (right-hand size).

employs the idiom on the left-hand side may find it easier to understand.

Recent work (Medeiros et al., 2019; Gopstein et al., 2017; Langhout and Aniche, 2021; de Oliveira et al., 2020) has attempted to elicit and catalog simple code patterns and language constructs that tend to confuse developers when reading them and for which there are less confusing, functionally-equivalent alternatives. These confusing constructs and patterns are called "atoms of confusion" (Gopstein et al., 2017) when it is possible to experimentally ascertain that there is a less confusing alternative (otherwise they are called "atom candidates"). The snippet on the left-hand side of Listing 1 contains two atoms of confusion discussed in previous work (Gopstein et al., 2017) focusing on the C language: Logic as Control Flow and Post-increment. Previous work has identified atoms of confusion in two languages, C and Java. According to the studies of Gopstein et al. (2017), removing the atoms Logic as Control Flow and Post-increment from small C code snippets improved the ability of study participants to predict their outcomes by 41% and 34%, respectively. For small Java code snippets, Langhout and Aniche (2021) report improvements of 53% and 46.27%, respectively. These results show that atoms of confusion may be **challenging** for developers using these languages. In addition, a study (Gopstein et al., 2018) with 14 large-scale open source projects written in C found that 4.38% of the lines of code have an atom and their presence has a strong correlation with bug fixing commits and long code comments. These results show that atoms of confusion are **prevalent** in large systems. They are also **relevant**, even for experienced software developers.

The goal of this paper is twofold. First, it aims to investigate whether atoms of confusion that were identified in Java and C also cause confusion to JavaScript developers. The latter, in spite of the syntactic similarities to Java, is a dynamically-typed language that is arguably more related to Scheme than it is to Java (Eich, 2018). Previous work has shown that, even for similar languages such as C# and Java, programmer behavior may differ significantly (Cabral and Marques, 2007; Ebad and Manzoor, 2016). JavaScript's programming culture differs from that of languages such as C and Java, particularly due to its dynamic capabilities and weaker type system. Furthermore, it includes elements such as array and object destructuring and spread that have no parallel in Java or C. At the same time, the language is not so different that many previously identified atoms cannot be represented in it. Second, the paper aims to verify whether other code patterns, not investigated in previous work, also tend to confuse JavaScript developers. Some of these code patterns are specific to JavaScript, e.g., object destructuring.

We present the results of a mixed-method research effort based on a repository mining study, two experiments, and a set of interviews. The repository mining study measures the prevalence of the atom candidates and helps us motivate the importance of studying these code patterns in more detail. We mined popular JavaScript projects from GitHub looking for the atom candidates that were investigated in both experiments. We found out that atom candidates are frequent and used intensively in 72 popular open-source JavaScript systems. The two experiments ask the participants to predict the output of small code snippets, where some of them have atom candidates and others do not. These two experiments were designed and conducted independently, by different researchers, with different samples, and complementary methodologies; one uses a repeated measures design (Keselman et al., 2001) whereas the other uses a Latin square design (George et al., 2005).

Both our experiments suggest that two code patterns that have been previously observed to confuse C programmers also confuse JavaScript programmers (with statistical significance, after correction): the comma operator and assignments being used as values. In addition, pre- and post-increment operators, when used as values, omitted curly braces, changes of literal encoding, and type conversions have led to confusion in at least one of the two experiments. For all these cases effect sizes were either medium or high. In addition, we have discovered that object destructuring and automatic semicolon insertion are potential JavaScript-specific atoms that have not been discussed in previous work.

To learn more about the misunderstandings in the snippets used in the experiments, we also interviewed 15 experienced professional developers. Participants preferred the code snippets without the atoms in 70% of the cases. The frequency with which the atoms appear *in the wild* might reveal cases where developers should be more or less careful when writing JavaScript code.

Artifacts availability. To enable the reproduction of our study, all tools, collected data, and Code Queries, Python, and R scripts are available online (https://github.com/rbonifacio/AtomsJS/tree/main/package).

2. Related work

The concept of program comprehension is central to software maintenance and feature development (Tilley et al., 1996; O'Hare and Troan, 1994). The study of program comprehension in the presence of atoms of confusion was introduced by Gopstein et al. (2017), who defined them as small code patterns that can verifiably lead to misunderstandings. In addition, for a code pattern to be considered an atom, there must be some alternative pattern or language construct that is functionally equivalent and less likely to cause confusion. One example that occurs in many programming languages is the Change of Literal Encoding. For instance, the C code statement `printf("%d", 013);` often leads programmers to predict the output to be 13, even though the correct answer is 11. This occurs because a leading 0 in a numerical literal indicates that the number is in base 8, a fact that is not only unknown to less experienced programmers, but misleading even for seasoned developers. After formulating a list of 19 atom candidates, Gopstein et al. (2017) were able to identify 15 code idioms and language constructs that present a statistically significant difference in comparative answer correctness when each of the 15 atoms was removed. The least confusing atom produced a 14% boost in prediction accuracy when it was removed, whereas the most confusing one showed a 60% accuracy increase. In a different work, Gopstein et al. (2018) presented the results of a comprehensive study on the incidence of atoms of confusion in the wild, considering open-source projects written in C and C++ programming languages.

These previous studies (Gopstein et al., 2017, 2018) have a great influence in our work. Even though JavaScript is a dynamic language that differs from C in a variety of aspects, it also has a number of constructs in common. In addition, the imperative aspects of the language are syntactically similar to C, e.g., assignments, conditional expressions, `if`-statements, pre and post increments and decrements, among others. On the one hand, this means that some of the atoms of confusion that exist in C programs may also occur in JavaScript code. On the other hand, differences between the languages and the surrounding programming cultures may lead to code patterns that are atoms of confusion in one language not being atoms in the other one. Compared to previous research, in this work we leverage the *Latin square* experimental design to control both participant experience and education better, while investigating the effect of the atom candidates in code comprehension. We present details about our setup in Section 3. Differently, previous research works (e.g., Gopstein et al. (2017) and Langhout and Aniche (2021)) use a *Randomized Partial Counterbalanced Design*, which does not *block* relevant characteristics of the participants.

de Oliveira et al. (2020) conducted an experiment using an eye-tracker with 30 participants involving three atoms in C. They found that code with atoms of confusion requires more time from participants to predict the output and more visual effort to comprehend. Gopstein et al. in another paper (Gopstein et al., 2020) further explored the original atoms by additionally conducting interviews and having programmers discuss among themselves the studied atoms. The authors argue in favor of complementing a quantitative analysis. Indeed, their study revealed findings such as a *"correct evaluation of an atom might not mean that a programmer understood its meaning"*. Here, we also complement the quantitative analysis with interviews.

Langhout and Aniche (2021) derived a set of 14 atoms for Java and performed an experiment with 132 students. For seven atom candidates (out of 14), they report that participants are more likely to make mistakes in the confusing version of the code. Castor (2018) presented a preliminary catalog of six atom candidates for the Swift programming language. Unlike JavaScript, in Swift, most of the atoms identified by Gopstein et al. (2017) are avoided by construction, e.g., it does not have assignments as values, increment operators, or macros.

Medeiros et al. (2019) analyzed 50 open source projects written in the C language with the goal of evaluating 12 code patterns called "misunderstanding patterns" by the authors. Many of these misunderstanding patterns are either atoms of confusion or atom candidates (Gopstein et al., 2017). This study shows that these patterns are prevalent; among these 50 projects, there are more than 109 K occurrences of misunderstanding patterns. In order to gauge the relevance of these misunderstanding patterns, the authors sent 35 pull requests removing occurrences of these patterns to randomly-selected open source projects. The authors of the study received feedback for 21 of these pull requests and the maintainers of the projects accepted 8 of them (22.86%).

3. Research method overview

The main goal of this research is to investigate the impact of atoms of confusion on JavaScript code comprehension. As such, in this paper we answer the following research questions:

(RQ1) What is the frequency of occurrence of atom candidates in practice (i.e., in open-source JavaScript projects)?
(RQ2) What is the impact of the analyzed code patterns on the comprehension of JavaScript code?
(RQ3) Do JavaScript developers identify atoms of confusion as contributing to program misunderstanding?

Answering RQ1 enables us to better motivate the other two questions by showing that studying these code patterns has practical relevance and discussing how often they occur in practice. Furthermore, by addressing research questions RQ2 and RQ3, we can either generalize or refute the findings about atoms of confusion already discussed in the literature. In addition, answering the third question allows us to enrich existing catalogs about atom candidates. Answering these research questions might also lay the foundations for the implementation of tools that can automatically transform code into cleaner versions.

We conduct a mixed-methods study to answer these questions. We perform a repository mining study (Section 4) that investigates the prevalence of some of these atom candidates in large scale, professionally-developed JavaScript software. This helps us motivate the remaining three studies. Our research also includes two independently designed and conducted experiments: a repeated measures/within subjects study (Section 5) and one using a Latin square, between subjects, counterbalancing experimental design to control for subject experience (Section 6). By using two different experiments, conceived by different researchers, with different designs, and conducted with different participants, we hope to rigorously verify the impact of atoms of confusion in code comprehension activities. Conducting multiple studies, with different designs and samples, is a typical approach in other areas of science to obtain more solid evidence and also to contest evidence obtained by single study, for example, Weggemans et al. (2001). We discuss this matter further in Section 8.3.

These two experiments explore the impact of atom candidates on understanding JavaScript code (goal of research question RQ2). They investigate whether or not programs that contain atom candidates tend to produce more misunderstanding for programmers trying to predict their outcome. We also conduct a set of interviews with developers (Section 7) to contrast the quantitative results of the experiments with their preferences and opinions (thus addressing RQ3).

Table 1 presents the atom candidates that we explore in all studies, while Table 2 presents the additional ones that only the repeated measures study analyzes.

4. Prevalence study

We mine a set of GitHub open source repositories and compute how often these atom candidates occur. We do this with the goal of quantifying the prevalence of the investigated atom candidates. If they rarely occur in real-world systems, this would suggest that they cannot be a significant source of confusion for developers. This study addresses our first research question (*What is the frequency of occurrence of atom candidates in practice?*).

4.1. Study setting

We first collect the most popular GitHub repositories that are primarily written in JavaScript. We measure popularity using the project's stargazers. This metric, available through the GitHub API, represents the number of stars a project received from users of the platform. The same metric has been used in a number of previous studies as a proxy to estimate projects' popularity (Gyimesi et al., 2019; Canedo et al., 2020). We then select the top 100 most popular repositories and remove projects that do not reach the first quartile of the distribution of lines of code (excluding from our dataset small projects).

After filtering out JavaScript project candidates, we build a curated dataset comprising a total of 72 repositories. Examples of projects in this dataset include React, Node JS, and AngularJS. Table 3 presents some statistics about the projects we consider in our research. The size of the projects range from small ones (5543 lines of code) to complex systems with more than 1 MLOC. All projects in our dataset have at least 1244 forks and at least

Table 1
Atom candidates analyzed in all studies. Nine of these atoms come from the original study of Gopstein and colleagues (Gopstein et al., 2017). For these atoms, both the descriptions and the examples, with the required adaptations, are taken directly from their paper.

Atom name	Description	Obfuscated	Clean
Logic as control flow	Traditionally, the && and −− operators are used for logical conjunction and disjunction, respectively. Due to short-circuiting, they can also be used for conditional execution.	V1 && F2();	if (V1) { F2() };

Assignment as value	The assignment expression changes the state of the program when it executes, however it also returns a value. When reading an assignment expression people may forget one of the two effects of the expression	V1 = V2 = 3	V2 = 3
			V1 = V2

Pre-Increment/Decrement	Similar to post-increment/decrement, these operators change a variable's value by one. In contrast to the other operators, pre-increment/decrement first update the variable then return the new value.	V1 = ++V2	V2 + = 1 V1 = V2
Post-Increment/Decrement	The post-increment and decrement operators change the value of their operands by 1 and return the original value. Confusion arises because the value of the expression is different from the value of the variable.	V1 = V2++	V1 = V2 V2 + = 1

Comma operator	The comma operator is used to sequence series of computations. Whether due to its eccentricity, or its odd precedence, the comma operator is commonly misinterpreted.	V3 = (V1 + = 1; V1)	V1 + = 1;
			V3 = V1;

Implicit predicate	The semantics of a predicate are easily mistaken. The most common example happens when assignment is used inside a predicate or when a success state is represented as 0.	if (4% 2)	if (4% 2 != 0)

Conditional operator	The conditional operator is the only ternary operator in C, and functions similarly to an if/else block. However, it is an expression for which the value is that of the executed branch.	V2 = (V1==3)?2:V2	if (V1 == 3) {
			V2 = 2;
			}

Automatic semicolon insertion	Semicolons in JavaScript are optional in some contexts. However, some cases where semicolons are optional can confuse the JavaScript parser. This may lead to surprising behavior.	let V1 = 9
		(console.log(V1))
			let V1 = 9;
			(console.log(V1));

Omitted curly braces	When control statements omit curly braces, the scope of their influence can be difficult to discern	if (V) F(); G();	if (V) {F();} G();
Arithmetic as logic	Arithmetic operators are capable of mimicking any predicate formulated with logical operators. Arithmetic, however, implies a non-boolean range, which may be confusing to a reader	(V1–3) ∗ (V2–4)	V1!=3 && V2!=4

23,672 stars. We automate all the steps to filter, clone, and collect the statistics from the repositories using Python scripts.

We mine the occurrence of atom candidates from the repositories in our curated dataset using source code queries that we specified using the Semmle's CodeQL language (de Moor et al., 2007). CodeQL is an object-oriented variant of the Datalog language (Rodriguez-Prieto and Ortin, 2020), and currently supports researchers and practitioners to query the source code of systems written in different languages (such as C++, Java, and JavaScript). All CodeQL queries deal with a specific atom candidate, and collect the exact location in the source code where the atoms appear. Fig. 1 shows an example of a CodeQL query that mine occurrences of the conditional operator.

Although most of our queries are simple, we have to circumvent a few corner cases to avoid false positives. For instance, we do not consider atom candidates the pre-increment occurrences that appear within `for statement` definitions. We automate the process of running the queries and exporting the results to a format that simplifies our analysis (and also the reproduction of this study). Finally, we compute some descriptive statistics to measure the prevalence of atoms of confusion in practice.

4.2. Results

In this section we present the results of the software repository mining effort, whose goal is to reveal how prevalent atom candidates are in open source JavaScript projects. We mined 72 open source JavaScript repositories to understand how often atoms arise in real software. This study focused on atoms that were investigated in previous research (Gopstein et al., 2017) and that we explored in the remaining studies of this research (detailed in Sections 5, 6, and 7).

Similarly to previous studies (Medeiros et al., 2019; Gopstein et al., 2018) which investigate the prevalence of atoms of confusion in open source C and C++ projects, we found that atoms of confusion frequently arise in JavaScript open source systems. For instance, the six most frequently found atoms occur in at least 80% of the projects. Considering the extremes, atom candidates Conditional Operator and Implicit Predicate were found in all repositories we mined (see examples in Figs. 2 and 3), while Comma Operator occurred in only 15% of them (as seen in Table 4). Fig. 4 shows an example of the use of the Comma Operator in the `styled-components` project.

Considering all JavaScript projects in our dataset, we found a total of 364,873 atom candidates, though four atoms are responsible for 92.97% of this total: Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces. The remaining atom candidates comprise 25,621 occurrences combined, 7.03% of the total number of occurrences. The Comma Operator and Arithmetic as Logic atoms are the ones that arise less frequently (0.001% in total with 232 and 165 occurrences, respectively).

We calculate the frequency of each atom per KLOC. The four atoms mentioned in the previous paragraph, Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces, occur frequently in the analyzed projects. All of them have more than 5 occurrences per KLOC on average. In particular, two atom candidates occur very frequently, Implicit Predicate (19.89 occurrences per KLOC) and Conditional Operator (10.16 occurrences per KLOC). On the other hand, Arithmetic as Logic and Comma Operator occur less than once for every 50 KLOC. Given the

Table 2
Atom candidates analyzed only in the repeated measures study. Six of these atoms come from the original study of Gopstein and colleagues (Gopstein et al., 2017). For these atoms, both the descriptions and the examples, with the required adaptations, are taken directly from their paper.

Atom name	Description	Obfuscated	Clean
Type conversion	JavaScript will often implicitly convert types when there is a mismatch. Sometimes this conversion also results in a different outcome than a reader expects	{} + 4	{}.toString() + (4).toString()

Repurposed variables	When a variable is used in different roles in a program, its current meaning can be difficult to follow	let n = 1;
		n = “OK”;
			let n = 1;
			let s = “OK”;

Change of literal encoding	All numbers are stored as binary, but for convenience we represent numbers in decimal, hex, or octal. Depending on the circumstance, certain representations are more understandable.	let V1 = 013 console.log(V1)	let V1 = 11 console.log(V1)
Infix Operator precedence	JavaScript has more than 40 operators[a], the majority of them infix, each in one of 17 precedence levels with either right or left associativity. Most programmers know only a functional subset of these rules.	0 && 1 −− 2	(0 && 1) −− 2
Constant variables	Constant variables are a layer of abstraction that emphasize a concept rather than a particular value itself. When trying to hand evaluate a piece of code having a layer of indirection can obscure the value of the data.	let V1 = 5 console.log(V1)	console.log(5)

Dead, unreachable, repeated	Redundant code is executed to no functional effect, but its appearance may imply that meaningful changes are being made	V1 = 1;
		V1 = 2;
			V1 = 2;

Lack of indentation, no braces	Block delimiters and indentation are very important to understand nested structures in programs. If block delimiters are missing and indentation is incorrect, developers may misunderstand the nested structure of the program.	if (V1)
		if (V2)
		V3 = V3 + 2;
		else
		V3 = V3 − 1;
			if (V1) {
			if (V2) { V3 = V3 + 2; }
			else { V3 = V3 − 1; }
			}

Lack of indentation, with braces	Similar to Lack of Indentation, no Braces, but the block delimiters are present. The lack of indentation still has the potential to cause confusion.	if (V1) {};
		V1 = V1 − 1;
			if (V1) {};
			V1 = V1 − 1;

Property access	Object properties can be accessed using the dot (“.”) notation used in most object- oriented languages and using notation similar to how elements in collections like dictionaries and maps are accessed in other languages. The latter is less popular (Alves et al., 2022) than the former. Lack of familiarity may lead to confusion.	let V1 = {P1:0, P2:5 }
		V1[‘P1’] = 9;
			let V2 = {P3:0, P4:5 }
			V2[‘P3’] = 9;

Arrow function	The usage of arrow function notation, instead of the more traditional JavaScript notations for anonymous functions.	let V1 = x => x * x * x;
		let V2 = V1(5);
			let V3 = (x) => {
			return x * x * x;
			}
			let V4 = V3(5);

Array spread	Spread facilitates copying, extracting elements and concatenating arrays. Some caveats exist as the copying is shallow so this can cause misunderstanding between programmers.	let V1 = [1,2];
		let V2 = [5];
		V3 = [...V1, ...V2]
			let V1 = [1,2];
			let V2 = [5];
			V3 = [V1[0], V1[1], V2[0]]

Object spread	Similar to the spread in arrays, but with slightly more complex semantics because it may lead to the values of properties being overwritten if properties with the same names exist in two objects combined by means of spread.	let V1 = {P1:5, P2:8,
		P3:13};
		let V2 = {...V1, P3:99};
			let V1 = {P1:5, P2:8,
			P3:13};
			let V2 = {P3:99};
			V2.P1 = V1.P1
			V2.P2 = V1.P2

Array destructuring	This feature provides an approach to perform a limited form of pattern matching based on the elements of an array. The syntax mixes the introduction of new variables with array literals and is arguably less explicit than directly initializing new variables with array elements.	let [V2,,,V4] = [1,2,3,4];
		console.log(V2, V4);
			let V1 = [1,2,3,4];
			console.log(V1[0], V1[3]);

Object destructuring	Similar to array destructuring, but uses the syntax associated with object literals to introduce new variables by matching their names with the names of properties of the destructured object. This is less explicit than initializing the new variables with the values of said properties.	let V1 = {P1: 5};
		let {P1, P2 = 20} = V1;
			let V1 = {P1: 5};
			let P1 = V1.P1, P2 = 20;

[a] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators.

frequency of these code patterns in open-source JavaScript programs, it is worth it to investigate their impact on understanding JavaScript code. This is the goal of the studies we present in the following sections.

5. Repeated measures study

The results of the repository mining study show that some of the atom candidates we aim to study occur very frequently. Taking that as a starting point, we conduct two experiments and an interview study to gauge their potential to cause confusion. Furthermore, since we are focusing on JavaScript code, we also analyze a number of atom candidates that are specific to this language in one of the experiments.

The first experiment we perform uses a repeated measures, or within-subject, design (Keselman et al., 2001). This experiment is inspired by the work of Gopstein et al. (2017), but targeting

```
import javascript

from ConditionalExpr e
where not e.getTopLevel().isMinified()
select e, e.getLocation().getStartLine() as Location,
       e.getLocation().getFile()
```

Fig. 1. CodeQL query that searches for conditional expressions.

```
function prependModifierMarker (symbol, name, dynamic) {
  return dynamic
  ? ("_p(" + name + ",\"" + symbol + "\")")
  : symbol + name // mark the event as captured
}

function concat (a, b) {
  return a ? b ? (a + ’ ’ + b) : a : (b || ’’)
}
```

Fig. 2. Examples of the Conditional Operator atom present in the `vue` project (urls: shorturl.at/bJPS4 and shorturl.at/tJOS8).

```
React.useEffect(() => {
  if (props.config.selectedLines) { // Implicit Predicate
    dispatch({
      type: ’MULTILINE’,
      selectedLines: props.config.selectedLines,
    })
  }
}, [props.config.selectedLines])
```

Fig. 3. Example of the Implicit Predicate atom present in the `carbon` project (url: shorturl.at/aloqU).

```
var props = ((_props = {}),
 (_props[SC_ATTR] = stringifyNames(names)),
 (_props[SC_VERSION_ATTR] = ’4.3.1’),
 _props);
```

Fig. 4. Example of the Comma Operator atom present in the `styled-components` project (url: shorturl.at/sCKM0).

Table 3
Some descriptive statistics about the projects used in the MSR study.

	Min.	Median	Mean	Max.
Lines of code	5,543	36,161.5	111,432.33	1,278,405
Num. of forks	1,244	6,078	8,906.24	68,849
Num. of contributors	6	285.5	533.44	4,047
Num. of stars	23,672	34,990	46,919.51	310,935

Table 4
Summary of atom candidate occurrences in our dataset.

Atom	Projects	Occurr./KLOC
Implicit Predicate	100%	19.89
Conditional Operator	100%	10.16
Omitted Curly Braces	91.67%	6.61
Post Increment	90.28%	5.62
Pre Increment	83.33%	1.04
Assignment as Value	81.94%	1.02
Logic as Control Flow	56.94%	0.95
Automatic Semicolon Insertion	33.33%	0.13
Arithmetic as Logic	23.61%	0.02
Comma Operator	15.28%	0.03

JavaScript instead of the C language, and including some methodological differences. It covers all the atom candidates analyzed in the work of Gopstein et al. that can be transposed to JavaScript and adds candidates that are specific to this language.

5.1. Experimental design

In this experiment, the control variable consists of a tiny program containing a single atom candidate and the treatment is a functionally-equivalent version that does not contain the atom. The main dependent variable is whether subjects are able to correctly determine the output of the programs. The other dependent variable is the time required to correctly vs. incorrectly determine the outputs, independently of the presence of atoms. We use the same null hypothesis as Gopstein et al.: *“code from both control and treatment groups can be hand-evaluated with equal accuracy”* and the alternate hypothesis that *“the existence of an atom of confusion causes more errors than other code in hand-evaluated outputs.”*. For each atom candidate, we built six programs, three versions including the atom candidate and three versions not including it.

To identify atom candidates, we start out by selecting every atom candidate from Gopstein's study that can also happen in JavaScript. That excludes atoms related to pointers and preprocessor directives. Furthermore, we conduct an informal search of JavaScript questions on StackOverflow about constructs that are not typical of other languages, such as array and object spread, solutions to Code Golf[1] problems, and code style guides[2][3] to identify new atom candidates. We leverage previous work (Castor, 2018) on the identification of atom candidates to guide us in this informal search. Overall, the repeated measures study analyzes 24 atoms, 15 from the original study and 9 that are new (see Tables 1 and 2).

For each atom candidate, we write six short programs, three including and three not including the atom candidate. The former are the control for the study, or the **obfuscated** versions of the programs. The alternative, atom-free, versions of the programs are the treatment, or **clean**, versions. Each subject is exposed to one obfuscated and one clean version of each atom, totaling 48 programs, and should determine their outputs. As mentioned before, we measure answer correctness and the time to answer questions correctly vs. incorrectly. We control for learning effects (Neely, 1991) in three manners. First, by having multiple obfuscated and clean versions for each atom and presenting only one of each per subject. Second, by presenting the programs in a random order. Third, by presenting obfuscated and clean versions corresponding to the same atom candidate with at least 11 other programs in between, in accordance to the original experimental protocol (Gopstein et al., 2017).

Study instrument. The experiment is conducted by means of a questionnaire made available as a web application. As part of this effort, we carried out an informal pilot whose main objectives were (i) to spot bugs in the application and in the data collection mechanism; (ii) to gain feedback from respondents about the user experience of the application; and (iii) to formulate an estimate about how long answering the questionnaire would take on average. Based on the feedback of the participants of the pilot, we present only one obfuscated and one clean version of each atom candidate, to reduce the total time of the study and potentially increase participation.

We organize the study in three sections. In the first one we present instructions and also a check button whose checking means users agree that all collected data will be used anonymously and solely for research purposes. The instructions explain how the study works and asks them to dedicate their attention to it. We stress to participants the importance of not using any aids during the experiment, such as online or console interpreters.

The second section presents the programs, one at a time. For each one, there is a text box where the answer should be written and a "Next" button. If the subject leaves the text box empty and presses "Next", this is treated as a wrong answer. To reduce the likelihood of a subject attempting to execute the programs, the web application verifies whether the subject tries to change tabs or windows and presents a pop-up message if that is the case. Furthermore, it presents the programs as images, instead of text, to demotivate respondents from resorting to external resources by copying and pasting the code into an interpreter. Upon submitting an answer for a particular program, the subject is automatically led to a similar page, containing the next program.

We do not provide feedback about the time subjects take to answer each question. We also do not tell them whether their answers are correct or not. This aims at avoiding introducing bias for future respondents. Since we posted information about the study on social media platforms, possible threats could have arisen if we gave respondents instant feedback.

Study audience. We posted the link to the questionnaire on the authors' social media platforms, on Brazilian CS departments' mailing lists, and on two programming subreddits. We explained our research purposes and asked developers to take part in the study only if they have some familiarity with JavaScript. We collected 70 responses.

5.2. Results

In this experiment, we investigate whether 24 atom candidates impact readability. We examine how the performance of the participants differ when predicting the outputs of the clean and obfuscated versions of small code snippets (Section 5.2.1). Based on those results we establish which candidates can actually be considered atoms of confusion. In addition, similarly to Gopstein et al. (2017), we verify whether there is a trade-off between correctness and time irrespective of the presence of atoms; we check if participants who correctly predict the outputs of the programs tend to do so more slowly than the ones that make incorrect predictions (Section 5.2.2).

This study had 70 participants. Out of these, 67 have received university-level training in programming, 4 hold a PhD degree, 7 hold a master's degree, and 11 hold a bachelor's degree. The median programming experience of the participants is four years (mean 6.5 years) and the median experience with JavaScript programming is one year (mean 2.5 years).

5.2.1. Correctness analysis

Exploratory Data Analysis. For each of the 24 atom candidates, the participants predicted the outputs of two versions, one clean (control) and one obfuscated (treatment), as per the definition of a repeated measures design (George et al., 2005). Thus, each participant examined 48 code snippets. We also measured the time required by the participants to provide their answers. For most cases, participants made more mistakes when attempting to predict the outcomes of the obfuscated versions.

Table 5 presents the number of correct predictions for the clean and obfuscated versions of each atom candidate, as well as the percentage difference. In only three cases the participants predicted results correctly more often for the obfuscated versions, Arithmetic as Logic, Logic as Control Flow, and Property Access. The former two were empirically validated as atoms of confusion in the study of Gopstein and colleagues (Gopstein et al., 2017). For seven of the atom candidates, the number of correct predictions for the clean versions is more than twice the corresponding number for the obfuscated versions. Some of these atoms are common programming patterns in JavaScript programs, such as pre- and post-increments and object destructuring. The difference was even greater for Type Conversion (+440%) and Change of Literal Encoding (+300%).

Statistical analysis. Since this study uses a within-subject design, the data is paired and dependent. Our goal is to verify whether there is a significant difference in the performance of the same participants when examining clean and obfuscated versions of programs. The analyzed data is also dichotomous, since each participant either correctly predicts the output of a program or does not. Considering this scenario, we employ McNemar's test (McNemar, 1947), which is aimed at *"judging the significance of the difference between correlated proportions"*.

The "McNemar test" column of Table 6 indicates the p-values for the tests. These p-values are compared against an alpha of

[1] https://codegolf.stackexchange.com/.

[2] AirBNB JavaScript Style Guide: https://github.com/airbnb/javascript.

[3] Google JavaScript Style Guide: https://google.github.io/styleguide/jsguide.html.

Table 5
Summary of the correctness analysis.

Atom	Obfuscated	Clean	Δ (%)
Type Conversion	5	27	+440
Change of Literal Encoding	9	36	+300
Comma Operator	8	29	+262
Object Destructuring	15	42	+180
Assignment As Value	18	43	+139
Pre-Increment	13	28	+115
Post-Increment	8	17	+112
Array Destructuring	15	28	+87
Lack of Indentation no Braces	30	50	+67
Omitted Curly Braces	35	51	+46
Object Spread	21	30	+43
Array Spread	22	31	+41
Automatic Semicolon Insertion	15	21	+40
Infix Operator Precedence	47	53	+13
Arrow Function	54	58	+7
Implicit Predicate	41	44	+7
Lack of Indentation With Braces	53	55	+4
Conditional Operator	60	62	+3
Constant Variables	59	61	+3
Dead Unreachable Repeated	63	64	+2
Repurposed Variables	23	23	0
Arithmetic as Logic	52	47	−10
Logic as Control Flow	33	28	−15
Property Access	51	42	−18

Table 6
Hypotheses testing (correctness). Using Bonferroni correction, a p-value is considered statistically significant if it is lower than 0.0021.

Atom candidate	McNemar test	Odds ratio
Object Destructuring	<**0.0001**[*]	∞
Type Conversion	<**0.0001**[*]	12.00
Change of Literal Encoding	<**0.0001**[*]	28.00
Comma Operator	<**0.0001**[*]	11.50
Lack of Indentation No Braces	<**0.0001**[*]	11.0
Assignment As Value	<**0.0001**[*]	9.30
Pre-Increment	**0.0003**[*]	16.00
Omitted Curly Braces	**0.0009**[*]	6.30
Array Destructuring	0.0023	7.50
Post-Increment	0.0225	5.50
Array Spread	0.0490	3.30
Object Spread	0.0636	2.80
Property Access	0.0636	2.80
Infix Operator Precedence	0.1796	2.50
Arrow Function	0.2891	3.00
Arithmetic As Logic	0.3323	1.80
Automatic Semicolon Insertion	0.3915	1.40
Logic As Control Flow	0.4421	1.50
Implicit Predicate	0.6900	1.30
Lack of Indentation With Braces	0.7266	1.70
Conditional Operator	0.7905	1.30
Dead Unreachable Repeated	1.0000	1.50
Repurposed Variables	1.0000	1.00

[*] Indicate a statistically significant difference.

Table 7
Time in seconds required by the participants to correctly and incorrectly predict the outcomes of the programs.

	Mean	Median	Std. Dev.
Correct	35.44	24.36	38.43
Incorrect	44.50	29.29	62.48

0.0021, after applying Bonferroni correction (0.05/24). One of the atom candidates, Constant Variables, is omitted from the table because, due to the very low number of mistakes, it is not possible to perform the test. The table shows that eight of the 24 analyzed atom candidates can be considered atoms of confusion based on the results of our study, Object Destructuring, Type Conversion, Change of Literal Encoding, Comma Operator, Lack of Indentation No Braces, Assignment as Value, Pre-Increment, and Omitted Curly Braces. Among these eight, only Object Destructuring is JavaScript-specific. Six of the other atoms have been previously reported for the C language (Gopstein et al., 2017) and six for Java (Langhout and Aniche, 2021). For some candidates that have been previously identified as atoms, there was not a statistically significant difference. This is the case for Repurposed Variables, Logic as Control Flow, Implicit Predicate, and Conditional Operator. We examine this in Section 8.

We leverage the odds ratio (OR) as a measure of effect size, to quantify the meaningfulness (Ellis, 2010) of these results. The odds ratio is an appropriate measure of effect size for scenarios where McNemar's test is applicable (Mangiafico, 2016). For example, the odds ratio for Type Conversion is 12. This suggests that the odds of a participant correctly predicting the output of a clean version while incorrectly predicting the output of the corresponding obfuscated version is 12 times higher than the odds of correctly predicting the output of the obfuscated version and incorrectly predicting the output of the correct one.

According to the thresholds established by Chen and colleagues (Chen et al., 2010) (OR = 1.68 small, 3.47 medium, and 6.71 large), for most of the atoms where there is a statistically significant difference the odds ratio can be considered high. The value of the odds ratio for Object Destructuring is ∞ because no participant correctly guessed the output for the obfuscated version while incorrectly guessing the output for the clean one, which results in a denominator of 0. Omitted Curly Braces exhibits an OR of 6.3, which is borderline between medium and high (Chen et al., 2010). Array Destructuring and Post-Increment are two atom candidates that deserve a special note. The difference in the performance of participants examining clean and obfuscated versions of these atoms was not statistically significant, after correction. Notwithstanding, effect sizes were large and medium, respectively. This is a result that hints at practical relevance in spite of the p-values (Ellis, 2010). Further investigation is required for these two atom candidates.

5.2.2. Time analysis

Exploratory Data Analysis. In this section we investigate whether there is a trade-off between correctness and time required to predict program outputs, similarly to the analysis performed by Gopstein et al. (2017). For this analysis, we consider only whether participants correctly predict the outputs of programs or not, independently of the presence or absence of atoms, and the time required to do so. The null hypothesis we investigate is that there is no significant difference in time between correct and incorrect predictions. Intuitively, we would expect correct responses to take longer, on the basis that participants would spend more time carefully analyzing the programs before providing an answer.

Table 7 presents the mean and median times required by the participants to provide answers. The data is based on the 1648 correct and 1572 incorrect answers collected in the study. Contrary to our intuition, the mean time required by participants to provide an incorrect answer is 25% greater than the time required to provide a correct one and the median time is 20% greater. The standard deviations suggest that there is great variation in the times required by different participants, for different code snippets. Notwithstanding, the standard deviation for incorrect answers is 62.58% greater when compared to the same statistic for correct answers.

Statistical Analysis. We employ the non-parametric Mann–Whitney U test, also known as Wilcoxon's rank-sum test, to verify the aforementioned null hypothesis, i.e., whether the times for correct and incorrect answers can be considered identical.

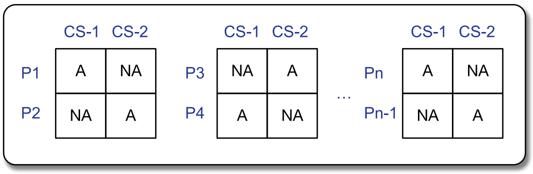

Fig. 5. Latin square design. Each "square" corresponds to a replica in our study. Each replica comprises two participants (square rows, e.g., P1 and P2) and two sets of programs (CS-1 and CS-2). We randomly apply the treatments (atom or non-atom code) to the cells of the squares.

That test produces a p-value of **0.00000046**, which shows that the difference is statistically significant and we can reject the null hypothesis. To gauge the magnitude of that difference we employ Cliff's Delta as a measure of effect size. This produces a result of **0.1027**, which suggests that, although participants tend to provide correct predictions more quickly, they do that only slightly so.

6. Latin square study

This study uses a different experimental design that makes it possible to control for the experience of the subjects. It employs a subset of the atom candidates used in the repeated measures study. More specifically, this study investigates ten of the atoms investigated in the repeated measures study: nine previously-validated atoms of confusion for the C language (Gopstein et al., 2017) that also exist in JavaScript programs plus one atom candidate that is specific to the JavaScript language (Automatic Semicolon Insertion).

6.1. Experimental design

The design of the Latin square study blocks two variables (subject experience and the programs) and considers two treatments: the presence or absence of atom candidates within the programs. To achieve such a design goal, controlling the effect of experience and individual programs, we resort to the *Latin Square Design* (George et al., 2005). Using this design we create a 2 × 2 matrix in which each row represents a subject and each column indicates the set of programs. The design of each square (a replica) is such that no treatment is repeated in the same row or column. For example, considering that we have a set of 10 code samples $s_0, s_1, \ldots, s_{10}$, if a given subject (P1) is asked to predict the output of the code samples $s_0, s_1, \ldots, s_5$ that contain atom candidates, then, when answering questions about clean programs, they will only be presented with clean versions of the code samples $s_6, s_7, \ldots, s_{10}$. Furthermore, a given subject (P2), who constitutes the second row of our example square, will be asked questions about the clean versions for $s_1, s_2, \ldots, s_5$, and will answer questions about obfuscated programs for $s_6, s_7, \ldots, s_{10}$. By doing that, we guarantee that all versions of the programs are contained within each square, and that each configuration occurs only once within a square. Fig. 5 offers a visual representation of the concept.

For each of the 10 selected atom candidates, we write one short program containing the atom. As in the repeated measures study, we call them the obfuscated versions of the programs. We also write corresponding short programs without the atom candidates and call them the clean versions of the programs. Overall, the experiment employed 20 programs, 10 obfuscated and 10 clean. In order to reduce the cognitive effort, each subject was asked to predict the output of 10 programs, five obfuscated and five clean. The order in which they are presented is randomized. By doing this, we seek to minimize the chances of subjects being aware that the current listing they are analyzing contains (or not) atom candidates. That is, each subject should indicate what would be the outcomes of the programs, some of them having obfuscated code (while other programs do not). Since participants are not exposed to obfuscated and clean versions corresponding to the same atom candidate, learning effect is not possible. We measure answer correctness and the total time each participant requires to participate in the experiment.

Study instrument. We implemented our experiment as a questionnaire in a web application and carried out an informal pilot. Undergraduate students and professional colleagues took part in the pilot. Some users reported layout defects, and many reported that the landing page did not explain the study well enough. We also spotted minor issues with our routines to create and populate the Latin squares.

We organize the questionnaire in three sections. The first section aims to characterize the subjects, asking their age, education level, and programming experience. We also include a check button, whose checking means users agree that all collected data will be used solely for research purposes. In the second section, we present to the participants a small set of instructions, where we explain how the study works and ask them to dedicate their attention to it. We stress to participants the importance of not using any aids during the experiment, such as online or console interpreters. For each question page, we kept track of whether or not the subjects switch windows.

The last section of the questionnaire presents a sequence of ten questions, each containing a program. For each question, there is a text box where the answer should be written. There is also an "I do not know" button, which, when clicked, leads the subject to the next question. In our setting, "I do not know" is treated as a wrong answer. The programs are also presented as images copied from a text editor. Upon submitting their answer for a particular question, the subjects are automatically led to a similar page, containing another program. Similarly to the repeated measures study, we do not provide feedback about time and correctness to the subjects.

Study audience. We posted the invitation to take part in the study on a JavaScript Reddit channel.[4] We explained our research purposes and asked developers of any level of expertise to take part. Within twelve hours, we collected more than 150 answers, populating more than 70 replicas of the Latin Squares. We collected significant data on time taken and discrepancies in answer correctness between obfuscated and clean versions of the programs. Some inconsistencies arose while building the squares, for instance, when a user quit in the middle of the questionnaire. We discarded from our study all squares that contained incomplete rows.

6.2. Results

In the Latin-square study, we estimate the impact of atoms considering the same two perspectives of the previous experiment: *correctness* (number of wrong answers) and *time* (how long to provide a correct answer). In particular, in the time analysis, differently from the repeated measures study, we compare the time required by participants to submit a correct response when analyzing code snippets with and without atom candidates. This study is based on the responses of 140 participants (a total of 70 replicas). All participants had taken at least some university course or hold a bachelor degree or equivalent. In addition, 21 participants hold a master's degree and three a doctorate degree. Considering the programming experience of our respondents, 19%

[4] https://www.reddit.com/r/javascript/.

Table 8
Summary of the correctness analysis.

Atom candidate	Obfuscated	Clean	Δ (%)
Comma Operator	28	65	+132
Automatic Semicolon Insertion	32	68	+112
Post-Increment	48	64	+33
Omitted Curly Braces	47	58	+23
Assignment as Value	56	68	+21
Implicit Predicate	58	68	+17
Logic as Control Flow	41	48	+17
Conditional Operator	60	66	+10
Pre-Increment	50	53	+6
Arithmetic as Logic	64	63	−2

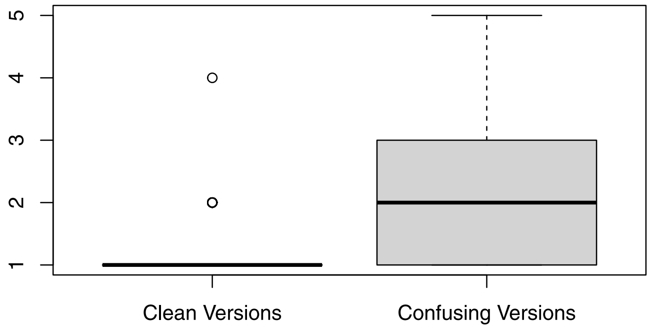

Fig. 6. Number of wrong answers of each subject.

have more than ten years of programming experience, 37% have between four and ten years of experience, 37% have between one and four years of experience, and 7% have less than one year of programming experience. Accordingly, we characterize the effect of atoms of confusion in JavaScript code taking into account the perceptions of both novice and experienced developers.

6.2.1. Correctness analysis

Exploratory Data Analysis. As mentioned in Section 6.1, each participant in this experiment evaluated ten code snippets, from which five were in their obfuscated versions, whilst the other five contained clean versions of the code snippets (i.e., without the atom candidates). As in the repeated measures study, the participants should provide the expected outcomes of the code snippets. Differently from that study, in this one each participant only predicts the outcome of one version of each code snippet, either clean or obfuscated. As discussed, we collect information about *correctness* (whether the participant correctly predicted the program's output) and *time*. Table 8 and Fig. 6 summarize the results of the correctness experiment.

Considering Table 8, the clean versions of six code snippets present at least a 15% improvement in answer correctness when compared with the obfuscated versions. In particular, the presence of the *Comma Operator* atom exhibits the highest impact on misunderstanding. Frequently used constructs and idioms, such as *Post Increment* and *Omitted Curly Braces* (see Section 4.2), also result in many mistakes. The boxplot of Fig. 6 shows a non-negligible decrease in the average number of incorrect answers when observing the clean versions of the code snippets. Also, the sample of responses with no atoms had almost no dispersion, which supports the argument that the clean versions of the code snippets are easier to evaluate correctly.

Statistical analysis. We first use the *Pearson's Chi-squared test* to investigate if there is a statistically significant difference in the frequency of correct and incorrect answers—due to the versions of the code snippets (obfuscated and clean code). The p-values for these tests are reported in the "Chi-square test" column of Table 10. For five atom candidates (Comma Operator, Automatic Semicolon Insertion, Post-Increment, Assignment as Value, Implicit Predicate), the results indicate that the obfuscated versions of the code snippets have a negative impact on code understanding (p-value $<$ 0.05). This result holds even after applying the Benjamini–Hochberg correction with a false discovery rate of 5%.

We measure the effect size of the clean version of the code snippets into the answers' correctness using the *Odds Ratio* (OR). The results are reported in the "Odds Ratio" column of Table 10. In the table, we also report the confidence interval (CI) for the OR in the "CI" column. Although many of the intervals are wide, for six of the atom candidates the lower bound of the confidence interval is greater than or equal to 1. This indicates that, at a 95% confidence level, a developer is likely to commit less mistakes when using the clean versions. For four of the atoms, Comma Operator, Automatic Semicolon Insertion, Assignment as Value, and Implicit Predicate, effect size can be considered large. For Implicit Predicate, it is medium. For instance, when comparing clean and obfuscated code snippets pertaining to the Comma Operator atom candidate, we observed an *Odds Ratio* of 19.02. This means that the odds of a correct answer are 19.02 times higher when interpreting the clean version—in comparison with the corresponding obfuscated version of the code snippet. Furthermore, with a 95% confidence level, the odds ratio is between 6.6 and 68.22, i.e., although the error margin is wide, it is strongly in favor of the clean version. If there is a significant likelihood of a developer committing a mistake when analyzing a clean version, the lower bound of the CI should be a number between 0 and 1 (since the OR is a ratio). This is the case for the four atom candidates in the lower part of the table.

Finally, we also use a *Binomial Generalized Logistic Regression* analysis to investigate if either *participant education* or *participant experience* impacts correctness. The findings suggest that *participant experience* impacts the results related to correctness (p-value < 0.001 and $\chi^2 = 24.18$). Fig. 7 summarizes the number of correct and incorrect answers over the four experience groups. Even though the presence of atoms of confusion reduces the number of correct answers for all *experience groups*, this effect is not uniform. For instance, novice developers (under one year of experience) provide a higher number of wrong answers for the obfuscated version of the code (57.8%), while developers with more than four years of experience provide more than 70% of correct answers even when evaluating the obfuscated version of the code snippets. It is also important to note that, independently of the *experience group*, developers tend to provide more than 84% of correct answers while predicting the outputs for the clean versions of the code. We replicate the *Pearson's Chi-squared test* for each experience group and find that the statistical difference is less significant for those developers with more than ten years of experience (p-value $= 0.007$ and $\chi^2 = 7.08$). Differently from *participant experience*, the factor *participant education* does not significantly change the correctness of the answers (p-value $= 0.236$ and $\chi^2 = 5.54$).

6.2.2. Time analysis

Exploratory Data Analysis. Table 9 shows the average time necessary for the participants to give a correct answer about the expected outcomes of a code snippet, considering both obfuscated and clean versions. We do not consider cases where the participants made mistakes. Seven atom candidates required more time from participants to predict a correct response. For these eight atom candidates, developers take at least 5.90% less time on average to find a correct answer when considering the clean version of a code snippet. In the extreme case (atom candidate Comma Operator), the participants take 76.23% less time on average to find the correct answer for the clean version of the

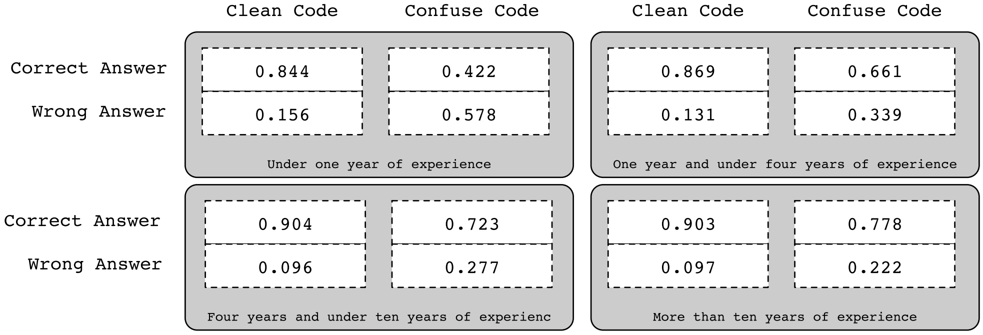

Fig. 7. Impact of atoms of confusion on the correctness (over the four participant experience groups).

Table 9
Time in seconds to submit a correct answer.

Atom	Obfuscated	Clean	Δ (%)
Comma Operator	87.67	20.84	−76.23
Automatic Semicolon Insertion	46.08	22.04	−52.17
Post-Increment	28.70	25.67	−10.56
Omitted Curly Braces	48.85	30.00	−38.58
Assignment as Value	52.47	48.95	−6.71
Implicit Predicate	36.24	24.01	−33.75
Logic as Control Flow	108.94	51.07	−53.12
Conditional Operator	41.80	39.34	−5.90
Pre-Increment	30.71	42.45	38.19
Arithmetic as Logic	28.82	37.20	29.06

code snippets. Not all atom candidates, though, require less time for the participants to predict the answer. In fact, for Arithmetic as Logic and Pre-Increment, the time to give a correct answer for the clean versions was 29.06% and 38.19% smaller than for the obfuscated versions—we did not confirm these atom candidates as atoms of confusion in the previous section.

Statistical analysis We use the *Mann–Whitney U test* to investigate the null hypothesis that developers spend the same amount of time to correctly predict the outcome of clean and obfuscated versions of a code snippet. The results of Table 10 suggest that we should refute the null hypotheses for five atom candidates (Comma Operator, Implicit Predicate, Logic as Control Flow, Pre-Increment, and Arithmetic as Logic), after applying the Benjamini–Hochberg correction with a false discovery rate of 5%. For the first three, the analysis suggests that the participants need more time to predict the outcome of the code snippet in the obfuscated code version. For the atom candidates Arithmetic as Logic and Pre-Increment, the participants take less time to predict the outcome of the code snippets in the obfuscated version. We also computed the effect size using Cliff's Delta statistic. In the table, negative effect sizes suggest that it took less time to correctly predict the output of clean versions of the snippets. Considering the thresholds established by Romano et al. (2006) ($0.147 < |d| < 0.33$ small, $|d| < 0.474$ medium, otherwise large), we found a large effect for the Comma Operator atom candidate; a medium effect size for Logic as Control Flow, and small effect sizes for Automatic Semicolon Insertion, Implicit Predicate, Arithmetic as Logic, Post-, and Pre-Increment.

7. Interview study

To complement the two experiments, we perform semi-structured interviews with professional JavaScript developers, aiming to identify their perceptions regarding programs containing atoms of confusion (research question RQ3). We also ask each participant if they know of any other JavaScript-specific construct or idiom they think is likely to make the code hard to understand. In this section we detail the protocol we followed to conduct the interviews and to analyze the results.

7.1. Participant selection

We invite the participants of the interviews using a snowballing technique. That is, starting from our network of contacts, we invite an initial set of candidates to take part in our experiments. From this initial list, we ask for an indication of additional candidates. Our main selection criterion is that all participants should have been working with JavaScript in their daily professional activities. We invited a total of 17 developers, and 15 of them agreed to participate.

7.2. Interview process

We conduct semi-structured interviews using web conferencing software. We record all the interviews with the consent of the participants. On average, the interviews last 26.29 min, with

Table 10
Hypotheses testing (misunderstanding and time).

Atom	Correctness analysis			Time analysis	
	Chi-square test	Odds ratio	Confidence interval	Mann-Whitey U test	Cliff's Delta
Comma Operator	<**0.0001***	19.02	(6.60, 68.22)	<**0.0001***	−0.53
Automatic Semicolon Insertion	<**0.0001***	39.33	(9.21, 356.62)	0.0980	−0.16
Post-Increment	**0.0015***	4.83	(1.73, 15.74)	0.1201	0.15
Omitted Curly Braces	0.0510	2.35	(1.00, 5.77)	0.9253	−0.01
Assignment as Value	**0.0035***	8.39	(1.81, 79.12)	0.8268	0.02
Implicit Predicate	**0.0112***	6.95	(1.46, 66.56)	**0.0029***	−0.29
Logic as Control Flow	0.2920	1.54	(0.73, 3.28)	<**0.0001***	−0.39
Conditional Operator	0.1590	2.73	(0.74, 12.57)	0.2407	−0.12
Pre-Increment	0.7015	1.25	(0.55, 2.85)	**0.0062***	0.27
Arithmetic as Logic	1	0.84	(0.22, 3.12)	**0.0172***	0.23

*Indicate a statistically significant difference.

Table 11
Demographic information of the participants.

ID	Education	JS experience	Other languages
P1	BSc degree	9 years	Java, PHP, C, Go
P2	HS degree	3 years	Python, Go, Dart, Lua, C++, C#
P3	BSc degree	4 years	Java, C
P4	Undergraduate	3 years	Python, C, C++, Java, Go
P5	Master student	3 years	Python, SQL
P6	Bsc degree	15 years	Java, PHP, C, Python, Ruby, C#
P7	BSc degree	6 years	C, C++, Java, Assembly, Kotlin
P8	PhD degree	2 years	Java, Python
P9	BSc degree	5 years	PHP
P10	Master student	4 years	Java, C# and Python
P11	Master student	4 years	Java, Erlang, C#, Cobol
P12	Master degree	1 year	C
P13	Master degree	13 years	Java, PHP
P14	Master student	3 years	C, Python, Ruby
P15	BSc degree	2 years	Java, PHP

the shortest one lasting 14.59 min and the longest one 43.06 min. Two of the researchers conduct the interviews, and a third one listens to all the recordings to cross-validate the collected data. The interviews have three main parts. In the first one, we ask the developers the following demographic information: name, email, gender, level of education, current job position, JavaScript experience in years, and other programming languages they have worked with. Table 11 summarizes this demographic information.

In the second part of the interview our aim is to allow the subjects to describe their JavaScript experience, as well as to allow them to reveal any JavaScript constructs they regard as innately confusing. This allows us to identify potential atom candidates that are more specific to the JavaScript language. Examples of questions we explore in this section include: *Does JavaScript favor developers to produce code that is hard to understand?* and *Do you regard any particular construct or idiom of the language as especially confusing?*

In the third part of the interview, participants are shown pairs of programs that were used in the experiments, where each pair contains a clean and an obfuscated version of the same atom candidate. For this part we only use atoms that appear in both experiments. The participants are asked to evaluate which version of the code is easier to understand. To avoid introducing bias in the answers, the interviewers do not explain that one of the versions in each pair contains the atom under investigation. Subjects are just presented the pairs of programs and allowed to take the necessary time to decide on the most readable one.

7.3. Interview analysis

We first transcribe each interview and then examine the broad distribution of the answers. Our goal is to build an initial understanding of the participants' perceptions with regards to the challenges to understand JavaScript code in general and JavaScript code with atoms of confusion in particular. We follow-up with an open-coding procedure, highlighting the main themes and quoting the answers of the participants. We present these results in Section 7.4.

7.4. Results

In this section we present the results of the interviews with the practitioners. We contrast the findings of the experiments of Sections 5 and 6 with the opinion of software developers about their preferences regarding the obfuscated or clean versions of the code snippets. For the interview study, we only considered atoms that were analyzed in both the repeated measures (Section 5) and the Latin square (Section 6) studies.

Table 12
Summary of participants' preferences for code snippets *with* and *without* atom candidates. The participants were only presented the code snippets, without any indication about whether one was confusing or not. Participants were also allowed to choose Neutral when they thought both sides were equally readable.

Atom	Preference (%)		
	Obfuscated	Clean	Neutral
Comma Operator	0	100	0
Automatic Semicolon Insertion	0	80	0
Post-Increment	20	73.33	6.67
Omitted Curly Braces	0	100	0
Assignment as Value	20	60	20
Implicit Predicate	20	73.33	6.67
Logic as Control Flow	20	60	20
Conditional Operator	60	26.67	13.3
Pre-Increment	40	46.67	13.33
Arithmetic as Logic	0	93.33	6.67
Overall	18	71.33	8.64

A total of 15 practitioners took part of the interviews. We collected information regarding programming experience, familiarity with JavaScript, and their opinion about the 10 atom candidates we explored in the both experiments. We first presented them pairs of clean and obfuscated code snippets and then asked them to discuss their preferences towards one version. We did not indicate in any way whether any version was assumed to be confusing or not.

7.4.1. The participants' perceptions of the atom candidates

Supporting the results of Sections 5.2 and 6.2, Table 12 shows that for eight out of the 10 scenarios surveyed, the majority of the respondents prefer the version of the code without the atom candidate. In no case the *neutral* ratio was higher than the option for the clean version. Only for the Conditional Operator atom candidate the participants preferred the obfuscated version, instead of the clean one. Fig. 8 shows the code snippets for the corresponding obfuscated and clean versions. For this atom candidate, some participants who preferred the left-hand side version (obfuscated) still believed that the right-hand side version (clean) was more readable. The following quotes were extracted from the transcripts with two interviewees:

> *"I prefer to write [code using the left-hand side version], but I think [the right-hand side version] is easier to read, especially for newer programmers".*

> *"When I am programming, I write code with the conditional operator, [...], but, to be honest, I still think that [the code using the right-hand side version] is easier to understand".*

The Pre-Increment atom candidate also caused a conflict in one of the interviewees, who regarded the clean version as simpler to understand, but would still opt to write code with the atom. In contrast, one of the participants found the version with the atom candidate more elegant, but recognized it was less readable, and was willing to sacrifice elegance for readability. In the repeated measures study, we found a significant difference in favor of the clean version, with a large effect size in its favor. In the Latin square study, no difference could be observed in terms of correctness. However, there was a significant difference in terms of the time required to predict the output correctly in favor of the obfuscated version.

When analyzing the Logic as Control Flow atom candidate, one of the interviewees gave an example of personal experience that might motivate one to avoid writing code using this construct:

```
let config = {size: 3, isActive: false};
const_config = config.isActive === true
             ? config
             : {size: 10};
console.log(_config.size);
```

Listing 2: *Left-hand side* (using the *Conditional Operator* atom).

```
let config = {size: 3, isActive: false}
let _config;
if(config.isActive === true) {
  _config = config;
}
else{
   _config = {size: 10};
}
console.log(_config.size);
```

Listing 3: *Right-hand side* (without the atom).

Fig. 8. Example of a pair of code snippets used in the interview pertaining to the *Conditional Operator* atom candidate.

> *"This one is interesting, because I have written code that looks like the left-hand side [obfuscated version], and my colleagues complained that it was difficult to understand. Nowadays I prefer to write code using the version on the right-hand side [clean version]".*

None of the two experiments we detailed in Sections 5 and 6 found a significant difference in correctness between clean and obfuscated versions of snippets related to the Logic as Control Flow atom candidate. However, in the Latin square study the participants analyzing clean versions of the snippets required significantly less time to predict a correct answer.

For two of the atom candidates, the interviewees were unanimous in their preference for the clean version: Comma Operator and Omitted Curly Braces. Not coincidentally, in both experiments there were significant differences in correctness in favor of the clean versions for both atoms. Regarding the first one, we could often notice during the interviews that the *left-hand side* (with the atom of confusion) caused significant confusion among the participants. One remark about the *Comma Operator* atom of confusion is listed below:

> *"The code in [the left-hand side version] is unlikely to be understood unless the programmer knows C or C++".*

As for the atom candidate Omitted Curly Braces, one of the interviewees mentioned that, although they understand why one would opt not to use braces for simple if-then-else statements, they still advised against it, on grounds that:

> *"[I prefer the right-hand side version of the code ...] If I want to see well-written, easily understandable code, then I also have to do my job. Therefore I believe that, since I do not know who is on the other end maintaining this code, and it could be any person with any level of expertise, then I try to write readable, easy-to-understand code".*

7.4.2. Confusion in JavaScript code

The final remarks drawn from the interviews are related to potentially confusing constructs suggested by the participants and their perspectives on JavaScript as a language. From the latter, we can also uncover some other forms in which the language itself might contribute to writing confusing code.

One of the participants mentioned that the use of JavaScript's prototype-based inheritance can make it difficult to understand code, particularly when involving deep prototype chains. When asked about particular JavaScript constructs or patterns that can make code difficult to understand, three participants cited the callback pattern – potentially leading to several levels of nested function calls – as extremely difficult to assimilate. One of the respondents stated:

> *"Nested callbacks are very confusing. Even writing them can be confusing, let alone understanding them."*

Two other developers implicitly touched upon the callback pattern, mentioning that it can be difficult to understand *asynchronous* programming in JavaScript. We also found other idioms that are JavaScript atom candidates, including *property access* via array subscription (`V1['P1']` instead of `V1.P1`) and *arrow functions* (see listings 4 and 5). The former was investigated in the repeated measures study but we did not find a significant difference between clean and obfuscated versions. As future work, we aim at investigating whether or not these atom candidates are more likely to introduce confusion in JavaScript code.

```
let inc = (x) => x + 1;
```

Listing 4: Example of arrow function.

```
let inc = function(x) {
  return x + 1
}
```

Listing 5: Alternative version without arrow function.

8. Discussion

In this section we discuss the results of the studies in combination.

8.1. Contrasting the results of the studies

The results of the multiple studies we have conducted, combined and contrasted with previous work on atoms of confusion, provide an in-depth perspective on which code patterns tend to lead to confusion and which ones may require additional investigation. Across different studies, targeting different languages and groups of participants, these patterns have produced confusion, even in constrained scenarios without other interfering factors. Existing studies judge whether a code pattern is an atom of confusion or not by evaluating whether participants incorrectly predict the behavior of programs including that pattern significantly more often than the behavior of functionally equivalent programs that do not include it. According to this criterion, there are five atoms of confusion that have been confirmed in the studies of Langhout and Aniche (2021), Gopstein and colleagues (Gopstein et al., 2017), and at least one of the experiments we conducted, **Type Conversion**, **Change of Literal Encoding**, **Omitted Curly Braces**, **Pre-Increment**, and **Post-Increment**. They are small code patterns that have been shown to cause confusion for different samples, in programs written in different languages (Java, C, and JavaScript). In addition, **Comma Operator**, and **Assignment as Value** have been shown to be atoms in both of our experiments and in the study of Gopstein et al. (2017). Two code patterns have caused confusion in the participants of one of our experiments and also in one prior study, **Indentation No Braces** (Langhout and Aniche, 2021) and **Implicit Predicate** (Gopstein et al., 2017).

Our interviews and the study of Langhout and Aniche (2021) have also analyzed the propensity of some of the atoms to induce confusion from a qualitative perspective, by asking developers about their preferences. In both cases, the obfuscated versions of **Omitted Curly Braces**, **Pre-Increment**, **Post-Increment**, **Implicit Predicate** were considered more confusing than their clean counterparts. Furthermore, the interview results are aligned with the experimental results: the clean versions were consistently considered less confusing than the obfuscated ones, except for one code pattern, Conditional Operator, which has not been confirmed as an atom by either of our experiments.

Considering the accumulated evidence for the aforementioned atoms, not only in terms of statistical significance but also effect sizes, it is reasonable to suggest that coding style guides avoid them if they aim to cater to a wide range of developer experience levels. These guides usually exist in contexts where multiple developers need to maintain the same codebase and they aim to promote uniformity[5] and readability.[6] Although personal preference and development experience have an impact on individual developers' ability to understand these code patterns, this is arguably not the audience that coding style guides should target.

In the repeated measures study, we identified one JavaScript-specific atom, **Object Destructuring**. In the Latin square study, **Automatic Semicolon Insertion** was identified as another JavaScript-specific atom. For both cases, p-values were much lower than the threshold and effect sizes were high. However, in the repeated measures study obfuscated versions of Automatic Semicolon Insertion were not significantly different from their clean counterparts. Our interviewees favored the use of semicolon over Automatic Semicolon Insertion. Finally, as discussed in Section 5.2.1, **Array Destructuring** exhibited a large effect size, in spite of the non-statistically significant difference (after correction). As discussed elsewhere (Ellis, 2010), this is a scenario that hints at practical relevance, especially considering the very low p-value (<0.003). Further studies investigating these code patterns, potentially using different response variables (Oliveira et al., 2020; Feitelson, 2021), are left for future work.

Some code patterns that have been identified as atoms in previous work were not confirmed in our experiments, at least from a correctness perspective. Comparing the results of the Latin square study to previous research, two atoms of confusion that Gopstein et al. (2017) confirmed for C and C++ do not lead to a statistically significant impact on correctness in our study, **Logic as Control Flow**, and **Conditional Operator**. **Conditional Operator** was the only atom candidate in our interviews whose obfuscated version was considered less confusing than the corresponding clean version.

Logic as Control Flow is a more interesting case. Considering the correctness perspective, none of our experiments has identified Logic as Control Flow as an atom of confusion. However, in the Latin square study there was a significant difference in the time required to make a correct prediction in favor of the clean version of this atom candidate, with a medium effect size. Furthermore, both Langhout and Aniche (2021) and Gopstein and colleagues (Gopstein et al., 2017) have identified it as an atom. Two other atoms have exhibited a statistically significant difference in time, in favor of their obfuscated versions (with a small effect size). One of them, **Pre-Increment**, has been confirmed as an atom of confusion in multiple experiments, i.e., participants are less confused by the clean versions. These conflicting results reinforce the importance of conducting experiments that consider multiple response variables (Oliveira et al., 2020). In addition, it is clear that factors such as the experience of the participants and the selection of code snippets can have considerable impact on the results of studies about code readability (Feitelson, 2021). Table 13 summarizes the correctness assessment results in our two experiments: Repeated Measures Study (column Exp01) and Latin Squares Study (column Exp02) and the experiments of Gopstein et al. (2017) and Langhout and Aniche (2021).

It is not feasible to combine the results of the statistical assessments coming from our two experiments—because their design differs significantly. Nonetheless, we can reason about the relationships between correct/wrong answers in both experiments and combine the results of the exploratory data analysis we presented in Sections 5.2.1 and 6.2.1. Considering the ten atoms explored in our two experiments, for six atom candidates the number of correct answers for the clean version of the code snippets is at least 25% higher than the number of correct answers for the obfuscated version of the code (see Table 14).

Besides that, the results of the two experiments reveal findings that diverge. That is, for six atom candidates (out of ten explored in both experiments), the results of the statistical tests differ. Three main reasons might explain the differences: (a) the code snippets used in the experiments are different, (b) the participants of the experiments are different, and (c) the design of the experiments and thus the analysis procedures are different. As such, additional research would be necessary to understand the impact of these items on the results of empirical assessments of atoms of confusion in general.

Our findings here suggest the need for replication studies in this field. Here we left an additional question: What would be the results of the original studies on atoms of confusion if an independent group reproduces the original research? Although the divergences in the two experiments might threaten our study's conclusions, we complement the experiments with a qualitative study based on interviews. We return to this issue of reporting the results of different experiments in Section 8.3.

[5] https://google.github.io/styleguide/.

[6] https://github.com/airbnb/javascript.

Table 13
Summary of the findings across the different studies. Here we show only the atom candidates that have been explored in at least one of our experiments. A **True** value means that the experiment confirms the code pattern as an atom candidate (**False** otherwise).

Atom Candidate	Gopstein et al. (2017)	Langhout and Aniche (2021)	Exp01	Exp02
Change of Literal Encoding	True	True	True	–
Assignment as Value	True	–	True	True
Logic as Control Flow	True	True	False	False
Post-Increment	True	True	False	True
Type Conversion	True	True	True	–
Conditional Operator	True	False	False	False
Infix Operator Precedence	True	False	False	–
Comma Operator	True	–	True	True
Pre-Increment	True	True	True	False
Implicit Predicate	True	–	False	True
Repurposed Variable	True	False	False	–
Omitted Curly Braces	True	True	True	False
Dead Unreachable Repeated	False	False	False	–
Arithmetic as Logic	False	False	True	False
Object Destructuring	–	–	True	–
Lack of Indentation no Braces	–	True	True	–
Array Destructuring	–	–	False	–
Array Spread	–	–	False	–
Object Spread	–	–	False	–
Property Access	–	–	False	–
Arrow Function	–	–	False	–
Automatic Semicolon Insertion	–	–	False	True
Lack of Indentation with Braces	–	–	False	–

Table 14
Results of the exploratory data analysis for the experiments (Repeated Measures and Latin Squares design). The Obfuscated and Clean columns show the number of correct answers for the obfuscated and clean versions of the code samples in both experiments. We compute the **total** number of correct answers for the obfuscated version of the code in both experiments (TO) and the **total** number of correct answers for the clean version of the code in both experiments (TC). The Δ column is given by $\Delta = \frac{TC-TO}{TO}$.

Atom candidate	Repeated measures		Latin square		TO	TC	Δ (%)
	Obfuscated	Clean	Obfuscated	Clean			
Comma Operator	8	29	28	65	36	94	161.11
Automatic Semicolon Insertion	15	21	32	68	47	89	89.36
Assignment as Value	18	43	56	68	74	111	50.00
Post-Increment	8	17	48	64	56	81	44.64
Omitted Curly Braces	35	51	47	58	82	109	32.92
Pre-Increment	13	28	50	53	63	81	28.57
Implicit Predicate	41	44	58	68	99	112	13.13
Conditional Operator	60	62	60	66	120	128	6.66
Logic as Control Flow	33	28	41	48	74	76	2.70
Arithmetic as Logic	52	47	64	63	116	110	−5.17

8.2. Addressing the research questions

The results of the first study (mining open source JavaScript repositories) provides evidence that, although atoms of confusion compromise program comprehension, they frequently appear in open source JavaScript projects. In particular, seven, out of 10 atoms considered in our study, appear in more than 50% of the 72 projects we analyzed. Furthermore, at least two of them are used intensively, more than once for every 200 lines of code. In summary, the first study allows us to answer the third research question.

Answer to RQ1: The MSR study reveals that several atom candidates explored in our research appear frequently in practice, and cleaning up the use of Post-Increment/Decrement and the Automatic Semicolon Insertion might improve the readability of JavaScript code substantially.

Altogether, the results of our experiments allow us to answer the second research question of this study as follows.

Answer to RQ2: Our experiments using repeated measures and the Latin square designs give evidence that some of the atom candidates in C and C++ programs that also exist in JavaScript correspond to a source of misunderstanding in JavaScript code. Two JavaScript-specific code patterns have been identified as atoms of confusion by at least one of our experiments: Object Destructuring and Automatic Semicolon Insertion.

The results of the interview study complement the understanding of atoms of confusion because the participants make clear the existence of a trade-off between code comprehension and other quality attributes. For instance, most of the participants prefer the version of the code with the Conditional Operator, even though they agree that its use might contribute to the misunderstanding of JavaScript code, particularly when novices are maintaining the codebase. The participants of the interview study also mentioned other possible sources of misunderstanding in JavaScript, including the use of prototype-based inheritance and nested call-backs (as discussed in Section 7.4). Other JavaScript atom candidates include Object Destructuring, Array Spread, Object Spread, and Type Conversion. In summary, the results of the second study (interviews) allow us to answer the second research question.

> **Answer to RQ3:** The qualitative analysis of the interviews supports the results of our experiments— JavaScript developers most often agree that atoms of confusion compromise source code understanding.

8.3. Why multiple studies?

It is not reasonable to consider something a scientific fact in an area as inexact as Software Engineering based on a single experiment. We present two independently-designed and conducted studies, with diverse code snippets, samples, and experimental designs, to provide stronger evidence. Due to the independence of the studies, the experiment using the Latin Square design does not cover all atom candidates the Repeated Measures experiment explores. Nonetheless, we believe that atom candidates confirmed to be atoms of confusion by both experiments arguably provide more reliable evidence of the potential to cause confusion of these code patterns for a wide range of developers, in diverse scenarios. That is the case for the Comma Operator and Assignment as Value atoms. Considering our work and others', there is little doubt that these are code patterns that lead to confusion. Furthermore, we complement the two experiments with interviews with 15 developers. Generally, in more mature areas of science, conducting multiple independent studies, potentially with different experimental designs, e.g., controlled vs. free-living experiments in the medical sciences (Weggemans et al., 2001), and examining their results in combination is the most reliable way of establishing truth.

The experimental designs we have employed in the two experiments have different trade-offs. The main advantages of having a repeated measures (or within subjects) study are the following:

+ They are better able to detect an effect size, given that one exists, since it is possible to collect data about both treatments from each participant.
+ There tends to be less variance between groups, since the same participant is subject to both treatments
+ Individual differences between participants are controlled for because each participant acts as their own control

Conversely, the main limitations of a repeated measures study are the following:

- Data is not independent. This precludes the use of some statistical techniques and tests.
- There is a potential learning/practice effect, since going through multiple examples may make the participants able to predict results of subsequent cases. This can be mitigated by separating examples pertaining to the same atom candidate, but not eliminated.
- Fatigue effect, since participants have to examine more code snippets. This was exacerbated in our case, due to the larger number of atom candidates. Although we could have mitigated this problem by limiting the number of atom candidates shown to the subjects, our decision was to ask the participants to evaluate pairs of code snippets for every atom candidate—and thus collecting a large number of answers.

The Latin square study is an example of a between subjects study. In a way, its main advantages are precisely the absence (or a reduction) of the disadvantages of the repeated measures study and its main disadvantages are related to the advantages of a repeated measures study:

+ Although there is some data dependency (since we collect different observations from a given subject), analyzing the data in the Latin square design allows us to isolate confounding variables better.
+ There is no possibility of learning/practice effect and the fatigue effect is reduced since each participant is only exposed to one version of each code snippet.

The main disadvantages of a between subjects study are the following:

- Statistical significance tends to be lower.
- Does not account for variance within groups.

The use of a Latin square mitigates some of the disadvantages intrinsic to within subjects design. For example, since less work is required from participants, it was possible to obtain more data for the Latin square study than we could get for the repeated measures study. This increases its statistical power, making it possible to detect smaller effect sizes.

In addition, to capture aspects that none of these studies can capture, e.g., personal impressions, we also leverage the interview study. Section 8.1 highlights that some atoms that are perceived as confusing did not cause participants of the experiments to commit mistakes, e.g., Arithmetic as Logic. This combination of three different studies, with different samples and designs provides a more in-depth perspective on the extent to which these code patterns confuse JavaScript developers than we could obtain by having a single study.

9. Threats to validity

In this section we discuss some of the main threats to the validity of this work.

Conclusion validity. In the context of our survey study, we apply different non-parametric statistical tests appropriate for the cases where data was categorical (correctness, Chi-square test of independence) and continuous (time, Mann–Whitney U test). Furthermore, besides reporting p-values, we also report effect size measures appropriate for each scenario (odds-ratio and Cliff's delta) and apply a p-value correction technique to avoid the multiple testing problem. Finally, it could be argued that the size of the samples is insufficient to make conclusions for some of the atoms, a common problem in Empirical Software Engineering. We estimate the sample size for each one of the atom candidates, considering that each one had a different number of samples (Table 8). To that end, we use the ϕ measure of effect size for each atom (based on the Chi-square statistic), the standard α coefficient of 0.05, and set the expected statistical power to 0.8, as usually employed in the literature (Ellis, 2010). We find out that the sample size is sufficient for all but one of the atoms where we had a statistically significant result for correctness, Implicit Predicate.

Construct validity. We use correctness as a proxy for program comprehension and predicted outcomes of small programs as a measure of correctness. As discussed elsewhere (Oliveira et al., 2020), this approach is a test of the developers' ability to trace programs. Although this is a common approach in studies about atoms of confusion (de Oliveira et al., 2020; Langhout and Aniche, 2021; Gopstein et al., 2017), other measures of correctness could have been employed, potentially yielding different results. As a complement to correctness in the survey, we have measured the time it took for the participants to correctly predict the outcome of the code snippets (either with or without atom candidates). Furthermore, we have asked the interviewees about their preferences when comparing confusing and clean versions of the programs.

```
import javascript

predicate hasTokenAfter(ReturnStmt r) {
  r.getLastToken().getNextToken().getValue() != ”}”
}


predicate isNextTokenVariable(ReturnStmt r) {
  r.getLastToken().getNextToken() instanceof IdentifierToken and
  r.getLastToken().getNextToken().getNextToken().getValue() != ”=”
}


predicate isNextTokenLiteral(ReturnStmt r) {
  ( r.getLastToken().getNextToken() instanceof StringLiteralToken or
    r.getLastToken().getNextToken() instanceof NumericLiteralToken ) and
  r.getLastToken().getNextToken().getNextToken().getValue() != ”=”
 }

from ReturnStmt r
where not r.getTopLevel().isMinified() and
 r.isSubjectToSemicolonInsertion() and
 r.getNumChildStmt() = 0 and
 r.getNumChildExpr() = 0 and
 not r.toString().matches(”%;”) and
 hasTokenAfter(r)
select r, r.getLocation().getStartLine(), r.getLocation().getFile()
```

Fig. 9. CodeQL query that searches for the atom Automatic Semicolon Insertion. Note that, for this query, we had to implement three CodeQL predicates.

Internal validity. Since the survey was conducted online with unknown participants, we have no way of confirming their levels of education and experience. We mitigate this threat by, in the data analysis, explicitly accounting for programmer experience and using an experimental design that allows us to isolate the impact of the treatment from factors such as experience and formal education level. Also, we did not have a way to prevent respondents from cheating, such as running the code on an interpreter, or consulting other people. We partially mitigate this threat by presenting images with source code, instead of text. This creates an obstacle for participants to run the code while taking the survey.

External validity. Our results suggest that the selected idioms and code constructs may lead to confusion for small code snippets, but it is not clear if that result extrapolates to other scenarios. Even though it is likely that in larger code bases the confusion induced by these constructs may be even greater, the existence of additional context may mitigate this effect. Another possible threat to the generalization of our results, in particular for the survey and interviews, lies in the fact that the analyzed atoms may rarely occur in real JavaScript code bases. To mitigate that threat, we have analyzed 72 popular open source JavaScript repositories and found out that most of them are common, occurring at least once per 1000 lines of code. Only Arithmetic as Logic and Comma Operator occur less frequently than once per 10,000 lines of code.

We manually devised sets of code snippets to use in our experiments. These code snippets should have been designed considering the concept of "minimality" (Gopstein et al., 2017) to avoid other sources of misunderstanding. After running the experiments, we observed that this concept was not well addressed in a few cases. In particular, the code snippet contained the atom candidate Logic as Control Flow in our second experiment (based on the Latin Square design) has additional atom candidates. This could have inflated the number of wrong answers in the obfuscated version of this code pattern. However, the experiment results show the contrary, and the findings of the second experiment do not confirm Logic as Control Flow as an atom of confusion. Further analysis should be conducted to understand the effect of multiple atoms in the same code snippet on program comprehension tasks.

Precision of our CodeQL queries. We relied on existing CodeQL JavaScript predicates to implement most of our source code queries. However, we still had to implement our own predicates for three specific queries: the queries that search for the atoms Automatic Semicolon Insertion, Comma Operator, and Omitted Curly Braces. Fig. 9 shows examples of predicates we have implemented. Our test cases and manual assessments revealed that our queries correctly identified the atoms in the source code. Unfortunately, we observed that a predefined CodeQL predicate (named mayHaveBooleanValue) we used to mine the Implicit Predicate atom might have produced many false positives. That is, there are situations where this predicate wrongly states that an expression does not have a boolean value. This issue might have inflated the number of Implicit Predicate occurrences we reported in Section 4.

10. Conclusions

This paper reports the results of a mixed-method research effort that allowed us to better understand the impact of atoms of confusion in JavaScript code. This involved a repository mining study, two experiments, and an interview study. The two independently-conducted experiments, which involved 210 JavaScript developers in total, asked them to predict the output of code snippets with and without atoms candidates. In at least one of the experiments, five atom candidates (Type Conversion, Change of Literal Encoding, Omitted Curly Braces, Pre-Increment, and Post-Increment) significantly impact program comprehension activities and can be considered atoms of confusion. Our efforts have two implications for practice. First, we present evidence that not all atoms of confusion validated to C, C++, or Java led to a statistically significant impact on program understanding for JavaScript. Besides that, our findings might be used to alert developers to avoid writing JavaScript code with certain atoms of confusion (e.g., Comma Operator, Automatic Semicolon Insertion, Post Increment/Decrement, Assignment as Value, and Implicit Predicate). Finally, our results might help tool developers to create program transformation tools for removing atoms that frequently appear in software. Our research also pointed out additional JavaScript constructs and idioms that might introduce misunderstanding. These include nested callbacks, *property access*, and *arrow functions*. As future work, we intend to reproduce our survey to validate whether or not these additional idioms truly affect the understanding of JavaScript code.

CRediT authorship contribution statement

Adriano Torres: Led all steps in the second study (experiment using Latin Square design), Contributed to the third study (interviews), Writing of the manuscript. **Caio Oliveira:** Led all steps in the second study (experiment using Latin Square design). Writing of the manuscript. **Márcio Okimoto:** Led all steps in the fourth study (MSR effort), Implementing the CodeQL queries, Writing of the manuscript. **Diego Marcílio:** Led all steps in the fourth study (MSR effort), Implementing the CodeQL queries, Writing of the manuscript. **Pedro Queiroga:** Led all steps in the first study (experiment using the Repeated Measures design), Writing of the manuscript. **Fernando Castor:** Led all steps in the first study (experiment using the Repeated Measures design), Writing of the manuscript. **Rodrigo Bonifácio:** Supervision, Co-supervised the whole research, Contributed to the design and execution of the second study. Writing of the manuscript. **Edna Dias Canedo:** Supervision, Co-supervised the whole research, Writing of the manuscript. **Márcio Ribeiro:** Co-supervised the whole research, Writing of the manuscript. **Eduardo Monteiro:** Co-supervised the whole research, Contributed to defining the statistical methods used in this research and implementing several R scripts for evaluating the outcomes of this research, Writing of the manuscript.

Declaration of competing interest

I declare that we have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared our experimental data as a replication package here: https://github.com/rbonifacio/AtomsJS/tree/main/package.

Acknowledgment

This research was partially supported by CNPq, research grant number 314834/2021-4.

References

Ajami, S., Woodbridge, Y., Feitelson, D.G., 2017. Syntax, predicates, idioms: what really affects code complexity? In: Proceedings of the 25th International Conference on Program Comprehension, ICPC. IEEE Computer Society, Buenos Aires, Argentina, pp. 66–76.

Alves, F., Oliveira, D., Madeiral, F., Castor, F., 2022. On the bug-proneness of structures inspired by functional programming in JavaScript projects. CoRR, abs/2206.08849 URL https://doi.org/10.48550/arXiv.2206.08849.

Cabral, B., Marques, P., 2007. Exception handling: A field study in Java and .NET. In: Ernst, E. (Ed.), ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. In: Lecture Notes in Computer Science, vol. 4609, Springer, pp. 151–175.

Canedo, E.D., Bonifácio, R., Okimoto, M.V., Serebrenik, A., Pinto, G., Monteiro, E., 2020. Work practices and perceptions from women core developers in OSS communities. In: ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020. ACM, pp. 26:1–26:11. http://dx.doi.org/10.1145/3382494.3410682.

Castor, F., 2018. Identifying confusing code in swift programs. In Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance 1 (6), 1–8.

Chen, H., Cohen, P., Chen, S., 2010. How big is a big odds ratio? Interpreting the magnitudes of odds ratios in epidemiological studies. Comm. Statist. Simulation Comput. 39 (4), 860–864.

de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J., 2007. .QL: Object-oriented queries made easy. In: Lämmel, R., Visser, J., ao Saraiva, J. (Eds.), Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. In: Lecture Notes in Computer Science, vol. 5235, Springer, pp. 78–133. http://dx.doi.org/10.1007/978-3-540-88643-3_3.

de Oliveira, B., Ribeiro, M., da Costa, J.A.S., Gheyi, R., Amaral, G., de Mello, R., Oliveira, A., Garcia, A., Bonifácio, R., Fonseca, B., 2020. Atoms of confusion: The eyes do not Lie. In: Proceedings of the 34th Brazilian Symposium on Software Engineering. SBES '20, Association for Computing Machinery, New York, NY, USA, pp. 243–252. http://dx.doi.org/10.1145/3422392.3422437.

Ebad, S.A., Manzoor, D., 2016. An empirical comparison of Java and C# programs in following naming conventions. Int. J. People Orient. Program. 5 (1), 39–60. http://dx.doi.org/10.4018/IJPOP.2016010103.

Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2021. An exploratory study on confusion in code reviews. Empir. Softw. Eng. 26 (1), 12.

Eich, B., 2018. A brief history of JavaScript. Available at https://www.youtube.com/watch?v=GxouWy-ZE80, Last access: July 25th 2022.

Ellis, P.D., 2010. The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results. Cambridge University Press.

Feitelson, D.G., 2021. Considerations and pitfalls in controlled experiments on code comprehension. In: 29th IEEE/ACM International Conference on Program Comprehension. IEEE, Madrid, Spain, pp. 106–117.

George, E., Hunter, W.G., Hunter, J.S., 2005. Statistics for Experimenters: Design, Innovation, and Discovery. Wiley.

Gopstein, D., Fayard, A.-L., Apel, S., Cappos, J., 2020. Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 605–616. http://dx.doi.org/10.1145/3368089.3409714.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M.K., Cappos, J., 2017. Understanding misunderstandings in source code. In: ESEC/SIGSOFT FSE. ACM, pp. 129–139. http://dx.doi.org/10.1145/3106237.3106264.

Gopstein, D., Zhou, H.H., Frankl, P.G., Cappos, J., 2018. Prevalence of confusing code in software projects: atoms of confusion in the wild. In: MSR. ACM, pp. 281–291. http://dx.doi.org/10.1145/3196398.3196432.

Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., Mesbah, A., 2019. Bugsjs: A benchmark of javascript bugs. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). IEEE, pp. 90–101.

Hermans, F., 2021. The Programmer's Brain: What Every Programmer Needs to Know About Cognition. Manning.

Keselman, H.J., Algina, J., Kowalchuk, R.K., 2001. The analysis of repeated measures designs: A review. Br. J. Math. Stat. Psychol. 54 (1), 1–20.

Langhout, C., Aniche, M., 2021. Atoms of confusion in java. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension. ICPC '21, New York, NY, USA.

Mangiafico, S.S., 2016. Summary and Analysis of Extension Program Evaluation in R. Rutgers Cooperative Extension, New Brunswick, Available at rcompanion.org/handbook/.

McNemar, Q., 1947. Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika 12, 153—157.

Medeiros, F., Lima, G., Amaral, G., Apel, S., Kästner, C., Ribeiro, M., Gheyi, R., 2019. An investigation of misunderstanding code patterns in C open-source software projects. Empir. Softw. Eng. 24 (4), 1693–1726.

Neely, J.H., 1991. Semantic priming effects in visual word recognition: A selective review of current findings and theories. In: Besner, D., Humphreys, G.W. (Eds.), Basic Processes in Reading: Visual Word Recognition. Lawrence Erlbaum Associates, Inc., pp. 264–336.

O'Hare, A.B., Troan, E.W., 1994. RE-analyzer: From source code to structured analysis. IBM Syst. J. 33 (1), 110–130.

Oliveira, D., Bruno, R., Madeiral, F., Castor, F., 2020. Evaluating code readability and legibility: An examination of human-centric studies. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. IEEE, Adelaide, Australia, pp. 348–359.

Rodriguez-Prieto, O., Ortin, F., 2020. An efficient and scalable platform for java source code analysis using overlaid graph representations (support material website).

Romano, J., Kromrey, J., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Proceedings of the Annual Meeting of the Florida Association of Institutional Research. pp. 1–3.

Tilley, S.R., Smith, D.B., Paul, S., 1996. Towards a framework for program understanding. In: WPC. IEEE Computer Society, p. 19. http://dx.doi.org/10.1109/WPC.1996.501117.

Weggemans, R.M., Zock, P.L., Katan, M.B., 2001. Dietary cholesterol from eggs increases the ratio of total cholesterol to high-density lipoprotein cholesterol in humans: a meta-analysis. Am. J. Clin. Nutr. 73 (5), 885–891.

Adriano Torres is a PhD student at The University of Adelaide, Australia. His alma mater is the University of Brasilia, where he completed his Bachelors and MsC. His research focuses on the development of methods and tools that automate the discovery and correction of software faults. He also has industry experience with distributed systems, and currently works as engineering lead for Luma Health Inc.

Caio Oliveira has a Computer Science Bachelor's Degree from the University of Brasilia. He is a front-end focused developer with nearly 7 years of experience with Angular. He has worked in projects as full-stack developer with technologies such as JavaEE, Django, Flask, Spring boot, Laravel, and all of them had Angular as the frontend technology. Has familiarity with Javascript and Typescript languages.

Márcio Okimoto is a Database Administrator and junior researcher. He is a specialist in Business Intelligence and Data Warehousing. He also has a Master's Degree in Computer Science from the University of Brasília (UnB) and his major research interest is the Mining Software Repositories (MSR) field with a focus on productivity.

Diego Marcílio is a Ph.D. student in the Software institute at USI Universitá della Svizzera italiana in Lugano, Switzerland. He received his M.S. in Informatics from the University of Brasília, Brazil, in 2019. His research interests center around aiding developers when understanding, writing, evolving, and maintaining source code.

Pedro Queiroga is a Computer Science undergrad student at Universidade Federal de Pernambuco and works as a software engineer for CESAR, in Recife. Pedro's interests include Functional Programming, Computer Science Education, Computer Science History and Free and Open Source Software.

Fernando Castor is an assistant professor at the Department of Information and Computing Sciences, Utrecht University, The Netherlands, working under prof. Sjaak Brinkkemper, and an associate professor (on leave) at the Informatics Center, Federal University of Pernambuco, Brazil. His research interests center around program understanding, energy consumption, and exception handling.

Rodrigo Bonifácio is an associate professor at the Computer Science Department, University of Brasília, Brazil, and a member of the Software Productivity Group, led by Prof. Paulo Borba. His research interests lie in understanding the common usage of programming language features, program transformations, software architecture and modularity, and software security.

Edna Dias Canedo is currently an associate professor at University of Brasilia (UnB - Brazil). She received her Ph.D. (2012) degree in Electrical Engineering from the University of Brasilia (UnB). Her current research interests include Software Engineering, Requirements Engineering, Gender Diversity, Software Systems, Empirical Software Engineering, and Usability.

Márcio Ribeiro is an associate professor at the Computing Institute, Federal University of Alagoas, Brazil. His research interests lie in program refactoring, program understanding, program testing, software modularity, and high configurable systems.

Eduardo Monteiro is an associate professor at the Statistics Department in Universidade de Brasília, Brazil, and focuses his research in statistical modeling and applications.