Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# On The Impact of Atoms of Confusion in JavaScript Code

Caio Oliveira - 14/0176713
Adriano Torres - 16/0047617

Undergraduate dissertation presented as partial requirement
to acquire a Major in Computer Science

Advisor
Prof. Dr. Rodrigo Bonifácio

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# On The Impact of Atoms of Confusion in JavaScript Code

Caio Oliveira - 14/0176713
Adriano Torres - 16/0047617

Undergraduate dissertation presented as partial requirement
to acquire a Major in Computer Science

Brasília, 01 July 2019

# Dedication

This work is dedicated to both authors' families, who have supported and nurtured our development through our entire lives.

# Acknowledgements

First and foremost, we would like to thank our advisor, professor Dr. Rodrigo Bonifácio de Almeida, for dedicating some of his scarce time to helping us organize our research. His contributions, both at the conceptual and practical levels, were paramount to the development of this work. We would probably be unable to implement any automated detection whatsoever, were it not for Rodrigo's assistance with JavaScript's grammatical intricacies.

Special thanks to our friends Diego Marcílio and Danilo Santos. Diego also provided invaluable advice and support regarding automatic code transformation. Danilo worked closely with Caio in presenting the research to fellow students, which enabled us to run an intermediary survey with students of professor Geraldo Filho, to whom we are also immensely grateful.

We also want to thank our boss and friend Dr. Marco Marinho, for he has always provided personal example of excellence and persistence. Your indirect contribution in teaching us about not giving up the pursuit of great work is a lesson that extends beyond just this dissertation. We are honoured to have you in our examination board.

Finally, we thank the Reddit JavaScript community, which promptly engaged in responding our final survey, providing us with almost four times as much data as we expected. You contributions, praises, criticisms and objections were of great value to us.

# Resumo

Um dos aspectos mais importantes para a engenharia de software manutenível é a preocupaçao com quão compreensível o código fonte é sob a perspectiva humana. Uma vez que ter que despender muito esforço cognitivo pode desencorajar aquele que está tentando se familiarizar com novos trechos de código, torna-se evidente a importância de escrever código que seja o mais simples possível de entender. Além disso, o desenvolvimento de ferramentas que automatizem o processo de reescrita de código que é difícil de entender pode não apenas poupar o tempo que o programador gastaria para entendê-lo, como também o tempo que seria gasto escrevendo uma versão mais simples. Neste trabalho, apresentamos a metodologia e os resultado de uma pesquisa que conduzimos com mais de 200 programadores de diversos níveis de experiência e educação, onde buscamos isolar alguns dos menores trechos de código em JavaScript que possam confundir programadores, trechos tais que chamamos de *átomos de confusão*. Após medirmos diferenças nas previsões que programadores fariam sobre as saídas trechos de código que continham ou não tais átomos, mostramos que determinados átomos tornam o código significativamente menos compreensível. Para concluir o trabalho, propomos o uso de uma ferramenta de metaprogramação para automaticamente detectar código que contenha átomos de confusão.

**Palavras-chave:** Átomos de Confusão, JavaScript, Transformação de Programas, Métodos Empíricos

# Abstract

One of the main aspects of engineering maintainable software is the concern with how understandable the code is from the human perspective. Since having to spend a lot of cognitive resources can be discouraging when familiarising with new blocks of code, it is important to write code that is as straightforward as possible to understand. Moreover, developing tools to automate the process of rewriting code that is difficult to understand can save not only the time that would be spent understanding the code, but also the time spent rewriting its simpler counterpart. In this paper, we present the methodology and the results of a survey conducted on over 200 programmers of different levels of experience, in which we sought out to isolate some of the smallest possible snippets of confusing code in JavaScript, known as *'atoms of confusion'*. Upon measuring the disparity in answer correctness between confusing and simplified pieces of code, as well as differences in time taken to predict the program's output, we showed that certain constructs make the code significantly harder to understand. To conclude the work, we propose the use of a metaprogramming tool to automatically detect confusing code.

**Keywords:** Atoms of Confusion, JavaScript, Program Transformation, Empirical Methods.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The source code of a program can be understood as a collection of symbols and expressions to be interpreted by a computer (or compiled to a lower level representation and then executed by a computer). Nonetheless, any definition that takes only this aspect into account is partially complete. While it is not possible to execute programs without writing them in a way that the machine is able to interpret, it is also very difficult to develop software that perform complex tasks if one does not care about how easily other programmers are going to understand the source code [2].

That means programming should, in various ways, be regarded as an act of communication. In this context, the programmer is often faced with a dilemma. When initially confronted with a problem they have to solve, the programmer will proceed to employ their cognition into developing the logic of the solution, whilst simultaneously having to worry about writing "syntactically correct" code for the machine to parse and translate. The problem is that, during the time spent solving the problem and communicating with the computer, developers often do not have enough cognitive resources to also make considerations about whether their code is understandable by other programmers. Therefore, it takes a change of rhythm to make the code easier to understand [3]. Not proceeding with such change of rhythm is one of the main reasons why "confusing" blocks and idioms arise and remain in source code.

## 1.1 Problem Statement

A direct consequence of the presence of these idioms in the code base is that it becomes ever more difficult to develop new features for the software. Since "The ability to understand pre-existing source code is one of the most important elements of a continuously successful software project" [4], it follows that, if one cannot properly understand what the code is doing, they are not going to be able to build upon its existing parts. Other consequences of the presence of confusing code include: (a) increased probability of bug introduction, due to a lack of proper understanding of what the program does; and (b) loss of internal quality measures, such as cohesion and coupling. All of these comprise Steve McConnel's Software's Primary Technical Imperative: Managing Complexity [5]. This phenomenon usually leads to a decaying architecture. All these situations, which usually compound each other, expose the necessity of developing recommendations, tools and techniques that help us avoid writing confusing code, identify potentially confusing code, and transform confusing code into clear code.

In this work, we present the results of an empirical investigation about the implications of using specific JavaScript constructs that might hinder program comprehension—the so called atoms of confusion [4]. In general, at least one of the following conditions is sufficient to make small blocks of code confusing:

- A single line contains many statements, whose order of execution is unclear;
- It is not straightforward to predict the flow of execution of the code;
- The language allows a certain syntax, whose semantics differs from what programmers would expect it to be;
- The construct is eccentric and/or rarely used

## 1.2 Research Goals

In this research, we aim to collect evidence to answer two research questions:

(RQ.1) Do code snippets that contain atoms of confusion produce a higher error rate than snippets where the atoms were removed?

(RQ.2) Do code snippets with atoms of confusion require programmers to take longer to predict their output?

Based on these results, we provide evidence that, although the atoms of confusion considered in our study do not take a much higher amount of time to predict the output of the code, their impact on program comprehension is, in almost all of our cases, highly significant.

Our motivation to detect the presence and impact of atoms of confusion is to lay the foundations for the implementation of libraries that can automatically transform code into its simpler versions. By doing this, we could achieve the following objectives:

- Reduce the time taken for a programmer to familiarize with a new code base;

- Reduce the probability that readers will misinterpret what the code does;
- Decrease the chance of programmers introducing bugs into the code due to poor comprehension of the program's behaviour;
- Freeing programmers from having to spend time manually detecting and fixing the atoms, therefore allowing them to spend time in more productive tasks

To fix each atom of confusion, we proceed similarly to [4], where the authors proposed code whose behaviour was clearer by:

- Breaking statements into multiple lines, so as to make their order clear;
- Introducing delimiters to visually separate blocks of code;
- Replaced cognitively taxing constructs by simpler ones;
- Removed unconventional and eccentric syntax.

We also contribute with a metaprogramming library developed using the Rascal-MPL tool to identify atoms of confusion in JavaScript code. Using this library, we are able to parse through large portions of code, automatically detecting and counting the occurrence of each of the atoms we define. This work lays the foundation for the future implementation of a library that automatically replaces each instance of an atom by their simplified version, whilst preserving behaviour.

## 1.3 How This Work is Structured

In Chapter 2, we briefly discuss the importance of program comprehension, as well as previous work done in defining and detecting atoms of confusion, as well as measuring their impact on code readability. We also discuss our motivation for choosing JavaScript for our work.

In Chapter 3, we define research questions that can help us in confirming of refuting the hypotheses we formulated in this chapter. We also define each of the atoms whose impact we are going to measure, as well as propose simplified versions for each of them. We also describe the methods we applied to collect and analyse data, as well as the results. The chapter finishes with conclusions drawn from analysing the data, as well as threats to validity.

Chapter 4 offers a brief introduction to the Rascal Metaprogramming Language, which we used to automatically detect atoms in well known open source JavaScript projects. We describe the conditions and constraints we imposed into the detection of the atoms. We conclude this preliminary step of atom detection in JavaScript, and explain how it can be extended to other languages. We also discuss topics for future work. We finish the chapter by describing the challenges we faced, and the limitations of our detection tool.

# Chapter 2

# Background and Related Work

In this chapter we present previous work that has been done in measuring the impact of atoms of confusion. We also briefly discuss why program comprehension is important, the programming language processing pipeline, and why we chose JavaScript to conduct our research.

In this work, we implement software to mine JavaScript projects' repositories in its initial phase, which is followed by a second one where we employ a metaprogramming language to automatically detect atoms of confusion. This makes it necessary to understand how programs that operate on other programs (i.e., meta programs) work. For this, we provide a brief introduction to the stages of language processing with which our research is concerned.

## 2.1 Program Comprehension

The concept of program comprehension is central to software maintenance and feature development [6]. Regardless of whether doing maintenance on legacy software, or adding features to it, programmers have to understand what the source code does before making any improvements. A natural consequence of this is that a programmer must also have in mind that other programmers (or their future selves) are going to have to spend time trying to understand the code being currently developed before doing any work themselves.

As program comprehension is regarded less as a systematic process than as an objective [7], there is no single approach or framework that is capable of yielding easily understandable programs. Factors such as problem domain, size and complexity of the code base, and available tools all vary significantly between different working settings. These aspects, combined with differences between individuals, which can include experience, working memory capacity, and other cognitive traits, make it clear that program comprehension should be approached in different ways in different contexts.

One of the ways in which programs can become less comprehensible is if programmers favour complicated syntactic constructs instead of using semantically equivalent versions - when these exist - allowed by the language or framework they are working with. Although this can be, to an extent, a personal matter, since different people might hold different opinions on whether a given syntax is difficult to understand, it is possible to design an experiment whose goal is to measure if there exist elements of code that are considerably harder to understand, leading readers of the code to frequently make incorrect predictions of its behaviour. In the following sections, we elaborate on on a more precise definition regarding complicated syntactic constructs (Atoms of Confusion) and on software language engineering approaches [1] [8] that could be used not only to find, but also to transform the source code to improve program comprehension.

on a more precise definition regarding complicated syntactic constructs (Atoms of Confusion) and on software language engineering approaches [] that could be used not only to find, but also to transform the source code to improve program comprehension.

## 2.2 Atoms of Confusion

The concept of atoms of confusion was introduced in [9], which defined them as small code patterns that can reliably cause misunderstanding in a programmer. Their work laid an empirical framework to identify such atoms. By having subjects evaluate code wherein the expected output relied exclusively on evaluating the result of previously known confusing programs that had been written in C/C++, and asking the participants to predict the output of a set of code snippets where the atoms had been removed, the authors were able to isolate and measure the impact of such confusing atoms. One main example that occurs in many programming languages is the *Change of Literal Encoding*. In C, the following snippet

```
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 number literal indicates that the number is in base 8, a fact that is not only unknown to less experienced programmers, but also easy to be forgotten even by seasoned developers.

After formulating a list of 19 potential atoms, the authors were able to identify 15 snippets that presented a statistically significant difference in comparative answer correctness when each of the 15 atoms was removed. The least confusing atom showed a 14% boost in prediction accuracy when the atom was removed, whereas the most confusing one showed a 60% accuracy increase. The authors also performed a qualitative analysis on the different nature of confusion amongst the atoms, noting that in some atoms, a particular incorrect answer was prevalent, whilst there was a distribution of incorrect ones in other atoms.

Work done in [4] presented the results of a deep research on atoms of confusion that can be found in the C language. Although JavaScript allows us to program in different paradigms from the one C follows, most of the JavaScript code found in popular repositories are written under the imperative paradigm. Also, both languages share many syntactical rules, which allows us to replicate some of the C atoms in JavaScript.

The aforementioned studies represent successful attempts to characterize the concept of atom of confusion, and measure their impact on C/C++ program comprehension. It is, thus, reasonable to assume that other languages might contain their own confusing syntax. In this work, we build upon the idea of atom of confusion to build a tool that helps automate the detection of seven of the atoms we are researching. In the first step, we replicate in JavaScript some of the atoms detected in previous works. Our selection criteria focuses on selecting the most confusing atoms in [4] that are applicable to our context. This means some confusing constructs are not considered, such as macros and pre-processor directives, out, as JavaScript does not support them. We also introduce an atom that exists in JavaScript but not in C/C++, or Java. As shown in this work, such atom makes programmers highly likely to misinterpret code. Besides choosing a different language to focus on, in this work we also propose a metaprogramming approach for automatically detecting — and, in the future, removing — atoms of confusion. Automatic detection for six of the atoms measured has already been implemented, and development will continue with the goal of making possible the automatic detection and transformation of all the atoms detected in the survey conducted.

## 2.3 Language Processing Background

In computer science, a language means a set of valid sentences. Formally speaking, we say that, if $A$ is the set of all strings accepted by a finite automaton $M$, then $L(M) = A$, meaning $L$ is the language of of $M$ [10]. For practical purposes, though, determining whether a sentence is valid is not just a matter of syntax. We must also devise procedures to decide about their semantics. Moreover, as programming languages evolved over time, changing from strings of 1's and 0's into structures that more closely resemble human language, there arose the need to develop algorithms and heuristics whose task is to translate high-level languages into lower-level ones. Addressing these issues is the concern of Language Processing.

### 2.3.1 The Language Processing Pipeline

As shown by Figure 2.1 [1], the general pipeline for processing a language has three main stages. In the first one, we scan the input code and attempt to recognize it. This process is commonly known as the parsing stage. If this scan succeeds, processor moves onto generating an *Intermediate Representation*, which can be useful for the next stage, in which semantic analysis is carried out. Should we need to translate the code into another language, such as when generating machine code from C source code, we would employ a code generator. We can also employ code generation to transform existing code in a given language into code in the same language. In our case, we make use of an intermediate representation called the Abstract Syntax Tree to detect and atoms of confusion, as well as to transform code that contains them. Figure 2.2 [11] displays the entire pipeline for an end to end compiler.

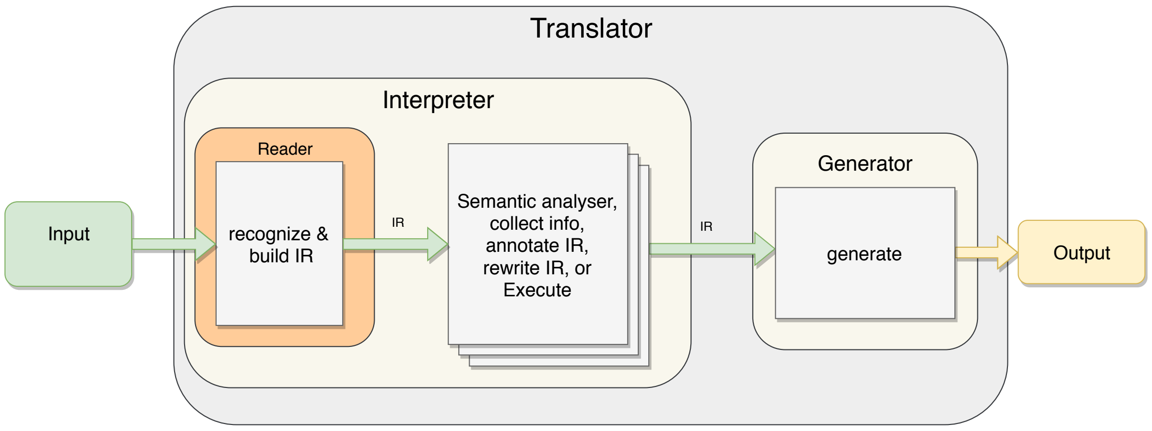

Figure 2.1: Multistage Pipeline

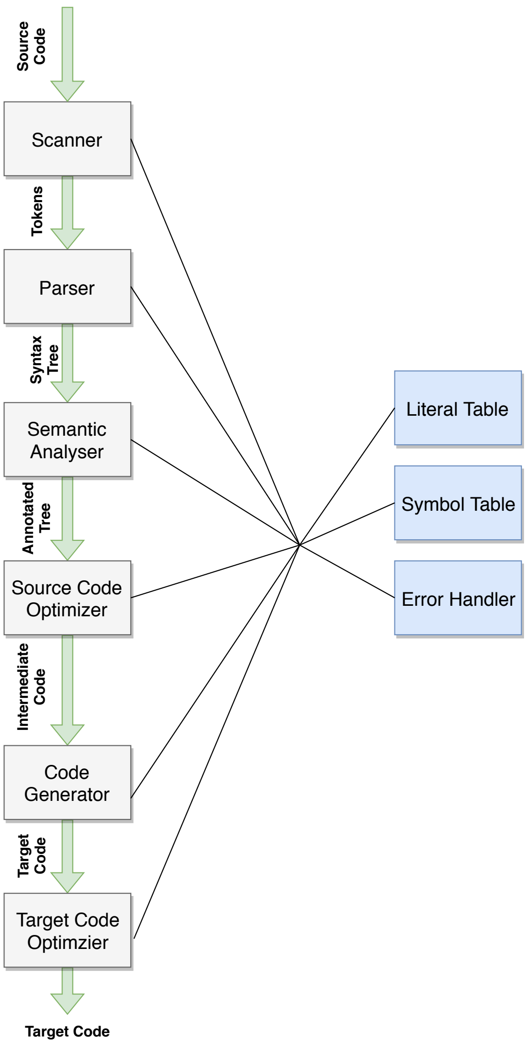

Figure 2.2: Complete pipeline for a compiler [1]

In the following sub-sections, we provide a high level presentation of the scanning and code transformation processes. As our tool is not concerned with semantic analysis, we skip the description of this step. For thorough discussions on language processing and compiling, refer to [11] and [1].

## 2.3.2 Scanning

The first step into the translation process consists in recognizing the input. This input entails symbols that are combined to create sets of statements and expressions. Therefore, recognizing the input in this context means deciding whether a certain sentence belongs to the language. In a similar way that what happens with natural languages, this recognition is usually carried out in two steps:

- Lexical analysis: In this step, the decision occurs at the word level. During this stage, the parser decides whether all the lexemes of the input phrase are valid within the language to be parsed. In an analogy with natural language, this is the equivalent to first identifying if all the words being recognized actually belong to the language in which the communication is happening. The elementary lexemes are called tokens.
- Syntactical analysis: This stage consists of deciding if the sentences assembled by the tokens are valid within the rules of the language. This is also similar to natural language, although it is more strict for programming languages. Ambiguity is common — in fact, inevitable — in spoken language, but this is not allowed for programming languages. All sentences must have a single exact meaning.

This process is carried out by *tokenizing* the input, and then performing some type of lookup to see if all the tokens actually belong to the language. If any token is not recognized, an error is thrown. If the process is successful, the parser moves on to syntactical analysis, which determines if the sentences uniquely match the language's grammatical rules. During this process, an intermediate data structure—usually a type of tree—is generated. A basic example of such representation is the *parse tree*, which contains tokens at its leaves, while the intermediary nodes represent the language's substructures, such as statements and expressions.

Figure 2.3 displays the parse tree that would be generated by any expression of the form A + B * C, provided that A, B and C are valid tokens. Note that, in contrast to natural languages, there can be no ambiguity as to what this means. In natural language, the sentence *'A plus B times C'* can be interpreted either as $A + B * C$ or $(A + B) * C$, but the same does not apply for a programming language. Such ambiguities are removed by predefined operator precedence hierarchies.

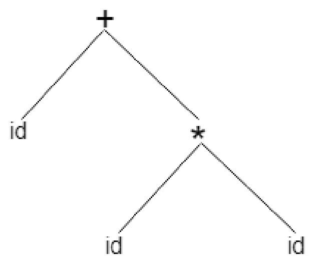

Figure 2.3: Parse tree of a simple arithmetic expression

What is important to conclude about this brief discussion of code scanning is that there must be a structure that defines the tokens and the syntactical rules of the language

— its grammar. Once a grammar has been defined, one can develop code to create and analyse another structure, namely the intermediate representation.

### 2.3.3 Code Transformation

As can be seen from figure 2.1, after building the intermediate representation, which is usually some type of tree, it is possible to rewrite some of its paths, therefore altering what the sentences mean, and, ultimately, what the program does. A discussion about how this is done is beyond the scope of this text, and can be found in texts about compilers, interpreters and programming languages.

In our context, we are interested in altering the intermediate representation whilst preserving behaviour. In other words, although we are going to to remove and/or insert nodes, and sometimes change the order they appear in the intermediate representation, we are not going to generate code whose sentences have a different meaning than the original code.

## 2.4 The JavaScript Programming Language

JavaScript is an object-based, interpreted language that was invented in 1995 by Brendan Eich. During that decade, web browsers were surging in popularity. As the web progressed to become a multi-application platform, JavaScript emerged as the natural candidate for client-side applications, as all the competing browsers of that time implemented support for the language. In the following years, there arose an organization, supported by Ecma International, whose goal was to standardize the different implementations of the language by the main browsers of the time, namely Netscape and Internet Explorer. This was the origin of ECMAscript, which is a specification and standardization for how the original language should be implemented. Therefore, rigorously speaking, when we talk about how browsers and engines implement JavaScript, we are really talking about their ECMAscript implementations. For the sake of clarity, in this text, we are going to use the terms JavaScript and ECMAscript interchangeably, as the differences between the two do not impact this work.

JavaScript is a single-threaded, prototype-based, multi-paradigm language, which means it supports object-oriented, imperative, and declarative programming. As it became ever more present in dynamic web pages, JavaScript became fundamental for web development. Due to its event-based architecture, in which events are queued for execution whenever the main execution stack is empty, JavaScript allows for asynchronous handling of events, such as mouse clicks or the page finishing loading. Such feature was paramount to it becoming so popular, since users would not experience blocking whilst interacting with the page. According to Stack Overflow 2019 Survey [12], JavaScript is the most popular technology in 2019. On top of that, it became the language of choice for popular web frameworks, such as jQuery and Node.js., making it one of the core languages of the world Wide Web Woday.

As in any Software Engineering project, JavaScript's development involved trade-offs between user flexibility and internal consistency of the language. While most of its regular syntax is very similar to those of Java and C/C++, the language has, as we will show, some unique and peculiar syntax. Such multitude of syntactic constructs, even though designed to simplify programmers' jobs, can be rather unclear. Moreover, such flexibility at times produce somewhat inconsistent semantic outcomes, which is exemplified by how the language deals with the *this* keyword. In this work, we set aside semantic considerations about the language, focusing on the syntactical constructions that can lead to poorly understandable code, as well as on how to write clearer versions.

## 2.5 The Atoms Analysed In This Work

As mentioned previously, our choice of atoms was primarily based on syntactical constructs in C that were also valid in JavaScript, with a particular atom that was exclusive to JS. In this section we briefly discuss each atom, and describe their potential for confusion. We also present an equivalent version of the code without the atom.

1. Arithmetic as Logic: We can represent logical statements in arithmetic form, which enables us to perform an arithmetic calculation instead of a Boolean comparison. For example,

   ```
   if(a !== 2 && b!==3) {
       console.log(a + b)
   }
   ```

   can also be written as

   ```
   if((a - 2) * (b-3)) {
       console.log(a + b)
   }
   ```

   since the multiplication will equal 0 only when a == 2 or b == 3. In JavaScript, the numeric value 0 will be coerced to Boolean false, causing the *then* block to be skipped. The potential confusion in this atom lies in the fact that programmers might assume that the operation will evaluate to true because it was successful, causing the condition to be true in all cases where the result is well defined. In this case, the non-confusing version of the code is to simply use Boolean comparisons instead of arithmetic when testing conditions.

2. Assignment as Value: Assignment operations always return a value. In JavaScript, an assignment returns the right-hand side of the operation. Furthermore, the assignment operation has right-to-left precedence. Both of these facts can be sources of confusion. When reading

   ```
   var1 = var2 = 6
   ```

   readers are likely to be unaware of the order in which the statements are executed. The most likely source of confusion lies in thinking that first var1 will receive the current value of var2, and then var2 will be assigned a new value of 6. What happens instead is that var2 is assigned a new value 6, which is also returned from the assignment operator and used to assign the same value of 6 to var1. The simplified version consists in breaking the statements into two lines, which makes the order of execution unambiguous:

   ```
   var2 = 6
   var1 = var2
   ```

3. Automatic Semicolon Insertion: In contrast to C and Java, where the insertion of the semicolon is mandatory at the and of most statements, JavaScript does not require its insertion. Instead, the interpreter will insert them into the code whenever it deems necessary. Compounded by the fact that JS syntax is very liberal with line breaks, the following code is valid, but its result is rather unclear:

   ```
    function calculate(input) { // Assuming numerical input for simplicity
    result = 10*input
    return
       result
    }
   ... // Using the value returned from the function
   value = calculate(1)
   if (value == 10) {
       console.log(value)
   }
   ```

   Since it is common practice in JavaScript to return object literals displayed across many lines, readers are likely to believe a Number object with 10 as its value will be returned. However, the interpreter is going to insert a semicolon after the return statement, ignoring the line break, and an *undefined* object will be returned. This is, in a way, unexpected, as JavaScript allows line breaks between many tokens. Furthermore, there will be no warning of invalid syntax. Therefore, *undefined* evaluates to false in JS, so there will be no visible output from the code above.

Assuming the programmer's intent is to return the actual value of the calculation, to remove the atom, we simply move the returned object to the same line as the return statement:

```
function calculate(input) { // Assuming numerical input for simplicity
   result = 10*input
   return result
}
```

Although subtle, this pattern dramatically decreases output prediction correctness, as we will see in the results section.

4. Comma Operator: This operator is used to sequence computation whose order would not be clear without it, such as in the following example.

```
result = (input--, input)
```

Although this operator is rather infrequent, and its syntax is not self-evident, its weird precedence can be very detrimental to the program's comprehension. What the operator does is it first executes in increment in input, and then it assigns the resulting value to result. Its unambiguous version is:

```
function calculate(input) { // Assuming numerical input for simplicity
   input--
   result = input
}
```

5. Ternary Operator: Like in C, the conditional ternary operator is the only one that takes three operands to produce an output. It works similarly to *if-then-else* statements, but it is an expression whose returned value is that of the executed branch. For instance,

```
canDrive = age >= 21 ? true : false
```

tests whether the age variable is greater than, or equal, to 21. If so, true is returned. Otherwise, canDrive is assigned false. The potential source of confusion in this case lies in the fact that the final value assigned to the canDrive variable is not as straightforward when the same logic is written using its *if-then-else* equivalent:

```
if (age >= 21) {
   canDrive = true
} else {
   canDrive = false
}
```

6. Implicit Predicate: Often present in conditional statements, this atom, as its name suggests, lies in assuming a certain predicate without making it explicit. Suppose we are trying to check an integer's parity. We could then write:

   ```
   if (a % 2) {
      console.log('Number is odd')
   }
   ```

   While technically correct, the code above does not make the intent of the calculation clear. In contexts where we have no previous knowledge about what condition is actually being tested, we might not be able to accurately predict the output. Also, similarly to the Arithmetic as Logic atom, one might assume that, if the modulo computation is successful, the condition will evaluate to true, leading the program to incorrectly interpreting any number as an odd number. To clarify the code, the suggestion is to make the condition being tested explicit:

   ```
   if (a % 2 !== 0) {
      console.log('Number is odd')
   }
   ```

7. Logic as Control Flow: In many programming languages, the || and && operators are used as disjunction and conjunction operators, respectively. JavaScript, like C, C++ and Java, implements left-to-right short-circuit evaluation. This means that when checking conditionals, if there is a conjunction operator, the first occurrence of a false operand will cause the ones to the right to not be checked, as the result is known to be false. This can sometimes be used to conditionally execute routines.

   ```
   canDrive && driveCar()
   ```

   In the code above, if canDrive evaluates to false, the driveCar() function will not be executed. In other words, driveCar() will only run if canDrive is true. If the reader is not aware of short-circuiting, or if they do not know the order in which disjunctions are evaluated, they might be led to think driveCar() will be called regardless of the value of *canDrive*. To make explicit the fact that we are controlling the flow of execution, we can use regular *if-then-else* statements:

```
if (canDrive) {
   driveCar()
} else {
   takeBus()
}
```

8. Omitted Curly Braces: *If* statements and *while* loops need no braces enclosing the following statement, if this a single line one. In this case, the statements can even be written in the same line:

   ```
   while (isHungry) eat(); drinkWater()
   ```

   Here, the eat() function is the trailing statement for the *while* loop, and *drinkWater* is executed when the loop is finished. By omitting the enclosing braces delimiting the statements executed by the *while* loop, we might lead readers to think *drinkWater* will also be called at each iteration. To improve readability, we break the code into different lines, and enclose the block executed by the *while* loop with braces:

   ```
   while (isHungry) {
      eat()
   }
   drinkWater()
   ```

9. Post Increment/Decrement: JavaScript inherited from C the inline increment and decrement operators, which allow us to change a variable's value in the same line as it is used in another operation. This may cause confusion, because the reader might not be sure about the order of execution. Consider the common pattern below.

   ```
   current = count++
   ```

   Although it should be clear to experienced programmers that the increment operator has higher precedence than the assignment operator, this might not be true to less experienced developers. Furthermore, a person reading a line that contains more than one statement might forego one of them, or have to spend more time to mentally figure out what the code does. Displaying the code in more than one line, making the sequence of statements more explicit, is a simple solution.

   ```
   current = count
   count = count + 1
   ```

10. Pre Increment/Decrement: Similarly to the previous one, the confusion is this atom arises from the fact that figuring out the order of the statements might not be immediate. In this case, the increment occurs before the assignment. It is not uncommon for readers to interpret both the Pre-Increment and Post-Increment operators as the same.

    ```
    current = ++count
    ```

    An analogous transformation is proposed for this atom, namely that of breaking the statements into multiple lines:

    ```
    count = count + 1
    current = count
    ```

The above definitions of our atoms are summarized in table 2.1. There is also a table at the end of the text where we display the actual code snippets we used in the survey, along with the expected output.

Table 2.1: Atoms of Confusion

| **Atom** | **Description** |
|---|---|
| Arithmetic as Logic | Use of the result of an arithmetic expression as control logic. |
| Assignment as value | Chained assignment statements |
| Automatic Semicolon Insertion | The JS parser automatically inserts semicolons, not requiring programmers to do so |
| Comma Operator | Used to sequence operations |
| Ternary Operator | Expression that returns a value after evaluating a condition |
| Implicit Predicate | The semantics of a predicate are not explicit in code |
| Logic as Control Flow | Use of short circuit evaluation to control flow of execution |
| Omitted Curly Braces & Indentation | Single *if, for or while* and without curly braces |
| Post Increment | The increment is done after all the operations |
| Pre Increment | The increment is done before any other operation |

In Chapter 3, we outline a methodology for conducting a survey to measure the impact of atoms of confusion on program comprehension. Then, in Chapter 4, we describe the employment of a metaprogramming language to automatically count the frequencies of the atoms we researched in Chapter 3.

# Chapter 3

# Impact of Atoms of Confusion on Code Comprehension

In this chapter, we describe how we proceed to measure the impact of 10 atoms of confusion which we hypothesise to impact program comprehension. We first discuss the research questions and study settings in Sections

## 3.1 Research Questions

The first objective of this research is to characterize the effect of atoms of confusion on JavaScript code comprehension. To achieve this goal, we conducted a survey, whose results should assist us in the investigation of the research questions we stated in Section 1.2.

By collecting answers from programmers with different levels of experience, we analyse the time subjects take to answer questions that contain atoms of confusion, and compare them with the time taken to predict the output of programs that behave the same way, but had the atom removed.

## 3.2 Study Settings

### 3.2.1 Research Instrument: A Survey

After choosing which atoms would be presented to the participants, we built a web application to collect their answers.

The first step of the survey was a quick profiling of the subjects, asking for their education level, age and programming experience. We also included a check button, whose checking meant users agreed that all collected data would be used solely for academic purposes. In the next page, participants were shown a brief instructions page, where we explained how the survey worked and asked them to dedicate their attention to it. We stressed to participants the importance of not using any aids during the survey, such as online or console interpreters. For each question page, we kept track of whether the subject had switched windows.

After this initial setup, the actual survey began. Each participant was shown a sequence of 10 questions, each containing a code block. For each question, there was a text box where the answer should be written. There was also an 'I do not know' button, which, when clicked, led the subject to the next question. In our setting, 'I do not know' was treated as a wrong answer. The code blocks were presented as images extracted from a text editor, so as to demotivate respondents from resorting to external resources by copying and pasting the code into an interpreter. Upon submitting their answer for a particular question, the subject was automatically led to a similar page, containing another snippet.

We decided not to provide feedback about the time students took to answer each question. Nor did we tell them whether their answers were correct or not. Our main concern was to avoid introducing bias for future respondents. Since we posted the survey in a social media platform, if we gave respondents instant feedback, they might post comments on particular atoms, therefore interfering with future future participants' thought process.

As we mentioned before, we first wrote the code listings in a text editor, and took pictures of it. In the case of an atom of confusion that was exclusive to JavaScript, which we called *Automatic Semicolon Insertion*, it was necessary to remove the syntax highlighter. Even though semicolons at the end of statements are optional to programmers in JavaScript, the interpreter automatically inserts them into the code. Our text editor was incorrectly highlighting a line break after a return statement, even thought it was valid JavaScript syntax. We had thus to turn the highlighter off to take the picture of this atom. We shall elaborate further on the topic in the following section, in which we describe the atoms we surveyed.

### 3.2.2 Latin Squares - Collecting And Evaluating Answers

Having selected the 10 atoms to be surveyed, we wrote the smallest possible code listings that contained each atom. We also wrote their functionally equivalent blocks with the atoms removed, which produced 20 snippets of code to be analysed by participants of the survey. In order to keep survey time low, we decided that each subject would be asked to predict the output of a subset of 10 listings, wherein each subset contained 5 blocks that contained atoms of confusion, whilst the remaining 5 had the atoms removed.

The order in which the questions were presented was randomized. By doing this, we were seeking to minimize the chances of subjects being aware that the current listing they were analysing contained (or not) atoms of confusion. In order to prevent the subjects from being biased by their previous answers, we needed a method of selecting questions in which, if a participant had answered a question that contained a specific atom of confusion, they would not be asked to predict the output of its non-confusing counterpart. For example, if a student was shown a block containing the Comma Operator atom, they would not be asked to analyse a snippet that had such atom removed. To attain such goal, we resorted to the *Latin Square Design* [13] [14]. In our setting, applying the design consists in creating a $2x2$ matrix in which each row represents a subject, and each column indicates the presence or absence of atoms of confusion in the subset they are assigned. The design of the square is such that no value is repeated in the same row or column. For example, if a subject A is asked to predict the output of listings that contain the atoms [1,2,3,4,5], then, when answering questions about non-confusing snippets, they will only be presented non-confusing versions of atoms [6,7,8,9,10]. Furthermore, student B, which constitutes the second row of our example square, will be asked questions about the non-confusing versions of [1,2,3,4,5], and will answer questions about confusing snippets of [6,7,8,9,10]. By doing this, we guarantee that all of our 20 snippets are contained within each square, and that each question occurs only once. Figure 3.1 offers a visual representation of the concept. Inconsistencies may arise when each square is being built. The main source of inconsistency we faced was when a user quit in the middle of the survey. When this happened, his row in the square was left incomplete. We considered all squares which contained incomplete rows to be invalid, and discarded them. Since we had a large enough number of samples, the squares we had to discard did not impact our results.

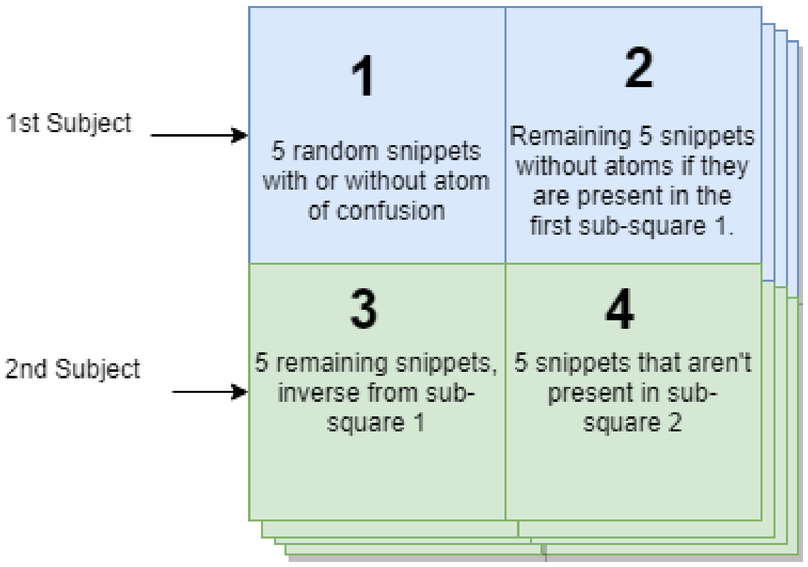
1st Subject
2nd Subject
1
5 random snippets with or without atom of confusion
2
Remaining 5 snippets without atoms if they are present in the first sub-square 1.
3
5 remaining snippets, inverse from sub-square 1
4
5 snippets that aren't present in sub-square 2

Figure 3.1: Latin Square

### 3.2.3 Survey Execution

#### Pilot Survey

To validate the web application we developed to conduct the survey, we ran an informal pilot survey whose main objectives were:

- To spot bugs in the application and in the data collection mechanism;
- To gain feedback from respondents about the user experience of the application;
- To formulate an estimate about how long the survey would, on average, take.

We had fellow undergraduate students, work colleagues and friends take the pilot survey. We discovered that the aspect that needed most improvement was the user experience. Some users reported layout defects, and many reported that the landing page did not explain the survey well enough. We also spotted minor issues with our routines to create and populate the Latin Squares. After performing all the necessary changes, we were ready to conduct another experiment.

#### A Survey With Undergraduate Students

Our second attempt was due to courtesy of a professor of our department. During the semester of the writing of this text, he was lecturing a Data Structures course, which is usually taken during the second semester of the undergraduate course. The professor agreed to take the students to a laboratory during one of his classes, and all the students who attended that day took the survey.

No issues were reported with the web application, and all answers were appropriately collected. This time, all the Latin Squares were being populated correctly, making the data we collected eligible for use in the final analysis. However, we opted to dispose of all the data we gathered in this occasion. What motivated this decision were the two following aspects:

1. The students were not focusing hard enough. Most students were not very serious about the survey. While we watched them answer the questions, we could see many students peeking at each other's screens. We could also clearly see students not taking nearly enough time to reason about the code, and answering the questions without giving it proper thought.

2. We needed a more diverse spectrum of respondents. In the beginning of our research, we established that we would need at least 25 Latin Squares in order to be able to have significant results. By conducting this survey, we were able to populate 11 Latin Squares. Compounded by the problem reported above, we were concerned that our spectrum of respondents might end up skewed towards undergraduate students with little or no JavaScript programming experience.

These two facts led us to conclude we needed to conduct a survey in which partakers engaged voluntarily, whilst also having an incentive to think hard enough about the problems they faced. To address these points, we prepared our final setting.

### An Open Survey Published On Reddit

In order to have subjects voluntarily sign up to take the survey, we concluded a laboratory setting might not be the best one. We then made a plan to establish contact with communities of JavaScript programmers on the Internet. Our initial plan was to try to engage contributors for two major JavaScript projects, namely Node.js and NPM. None of the projects, though, offered a direct way to interact with the community, such as an online discussion forums or mailing list. We would be required to first make contact with the projects' leaders, and only then would we have a chance to approach potential respondents. Due to time constraints, we decided to look for respondents elsewhere. We decided to post the survey on Reddit.[1]. We explained our academic purposes, and asked developers of any level of expertise to take the survey. To incentivize serious engagement, we proposed to raffle $50 gift cards on Amazon products at the end of the survey. Within twelve hours we collected more than 150 answers, populating more than 70 valid Latin

[1]Reddit is a North American online discussion platform. Its discussion threads, often called subreddits, are sorted by subject. For our research, we posted information and links to the survey in two JavaScript subreddits.

Squares. We were able to collect significant data on time taken and discrepancies in answer correctness between confusing and non-confusing versions of the snippets. In the following section, we present and discuss the results we obtained. We also elicit potential downfalls and threats to validity.

## 3.3 Results

In this section, we present the results of the final survey we conducted, and use them to attempt to answer the two research questions introduced in Chapter 3.

### 3.3.1 Data Collected From Participants

For our final survey, the following data about the subjects were collected:

- Education level;
- Years of programming experience.

Our initial plan consisted in also collecting participants email addresses for a raffle of gift cards on online store Amazon. However, many subjects objected this decision, claiming that, either they did not feel comfortable sharing their email address, or that they did not care about the gift and were only interested in contributing to the research. These two compelling arguments were convincing to us, and we thus did not collect email information.

### 3.3.2 Education Level and Experience

The following histograms describe the distributions of the subjects, according to their education level and years of programming experience, respectively.

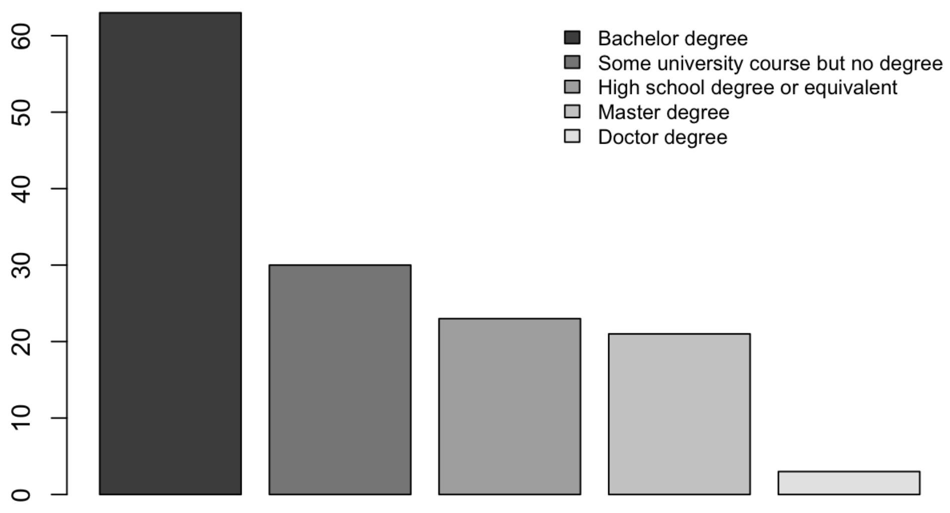

Figure 3.2: Participants' Education Level

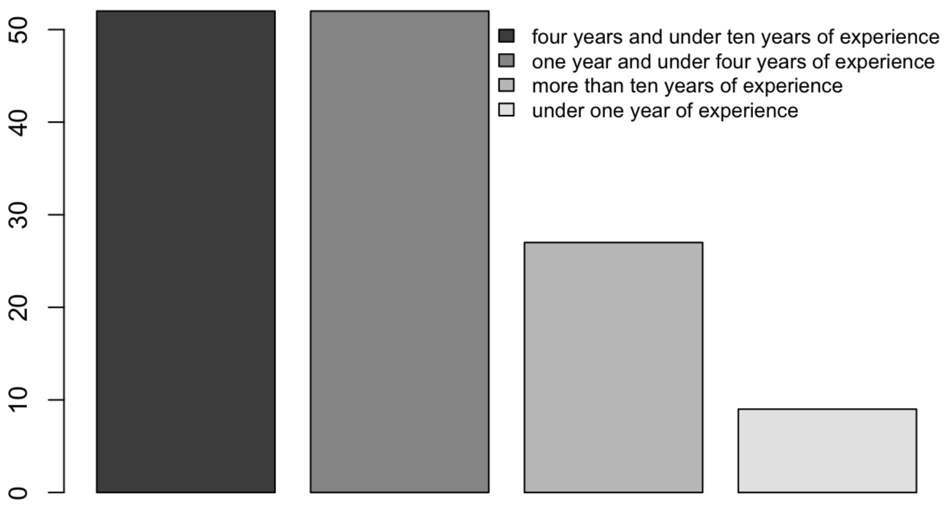

Figure 3.3: Participants' Experience

We can see in Figure 3.3 that more than half of the participants were either Bachelors, or had taken some university course, meaning the participants have had some level of formal training on programming. No subject reported to have never attended university.

More importantly, the second plot on Figure 3.2 shows that almost 93% of the participants have more than one year of experience with programming. About 37% of them have between four years and ten years, whilst another 37% comprises programmers with 1-4 years of experience.

Although we did not directly correlate a subject's level of experience with their answer's correctness, this data is important in that it supports the claim that, if experienced programmers, on average, commit more mistakes when answering questions about the snippets we defined as atoms of confusion, then a code base in which these atoms have been removed should be more readable and maintainable.

### 3.3.3 Data collected About The Subjects' Answers

As outlined in Chapter 3, each participant answered 10 questions, from which 5 were in their confusing versions, whilst the other 5 contained simplified versions. For each question, the following metrics were collected:

- Answer correctness: Whether the participant correctly predicted the program's output. Whenever the subject responded 'I do not know', the answer was considered incorrect;
- Time taken to answer the question;
- A flag to identify if the current question contained an atom;
- An identifier to link the question to the corresponding atom of confusion whose impact we were measuring;
- The IP address of the participants machine, used for the sole purpose of preventing a subject from instantly trying to answer the survey again from the same machine.

We also collected, for each question, a Boolean value that indicated whether participants switched windows during the questions. Our preliminary motivation for this was to measure if participants would be more prone to cheat, by opening up a console and running the code, in long and (presumably) confusing questions. However, we realised it would not be possible to control levels of honesty in an online survey. Also, as the results below will show, the differences in time were not in alignment with our hypothesis. In fact, in half of the atoms analysed, the time taken was equal, or even *greater*, when respondents were answering about the simplified versions. Taking these arguments into consideration, we decided not to analyse whether participants had switched windows during the survey.

Our final analysis consisted of 70 complete Latin Squares, which means 140 people responded an aggregate of 1400 questions, or 140 questions per atom, of which 70 contained confusing versions, whereas the other 70 contained simplified versions.

### 3.3.4 Answer Correctness

Table 3.1 summarizes the results of the analysed answers. We can see that, of the 10 atoms, 7 displayed a 15% or greater improvement in answer correctness. Unsurprisingly, the two eccentric constructs, namely the Comma Operator and the Automatic Semicolon Insertion, presented the highest discrepancies. But although these present the highest error rates, they are not the most expressive results. As we shall see in Chapter 5, we do not expect these atoms to occur frequently in production code bases. More important is the fact that frequent constructs, such as Post Increment and Omitted Curly Braces, represent significant degrees of confusion.

As the box plot in Figure 3.4 shows, there was a considerable decrease in the average number of incorrect answers when the atoms were not present. Also, the sample of answers where there was no atoms had almost no dispersion, which is a sign that the non-confusing code is easier to evaluate correctly.

Table 3.1: Difference in answer correctness between confusing and non-confusing pairs

| **Atom** | **%Correct** | **%Correct** | |
|---|---|---|---|
| | With AOC | Without AOC | $\Delta Correct$ |
| Comma Operator | 40 | 93 | 1.32 |
| Automatic Semicolon Insertion | 46 | 97 | 1.10 |
| Post Increment | 69 | 91 | 0.31 |
| Omitted Curly Braces | 67 | 83 | 0.23 |
| Assignment as Value | 80 | 97 | 0.21 |
| Implicit Predicate | 83 | 97 | 0.16 |
| Logic as Control Flow | 59 | 68 | 0.15 |
| Ternary Operator | 86 | 94 | 0.09 |
| Pre-Increment | 71 | 76 | 0.07 |
| Arithmetic as Logic | 91 | 90 | -0.01 |

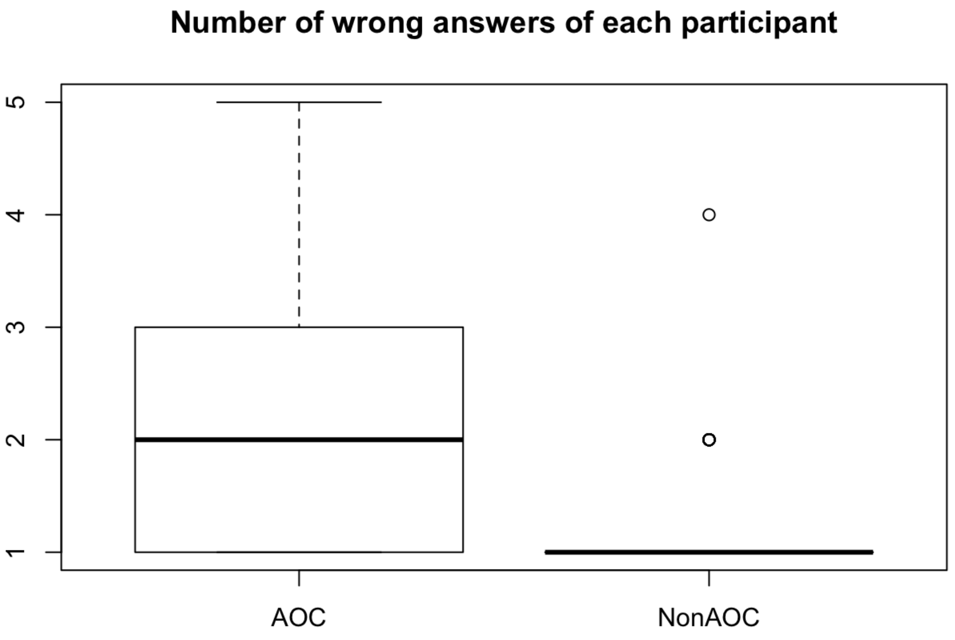

Figure 3.4: Number of wrong answers of each subject

We also did a logistic regression model, mostly because the analysis of a code snippet might lead either to a correct or wrong answer. In R, we can use the glm function to create a logistic regression model. Considering a confidence interval p-value $< 0.05$, two factors are relevant to lead to wrong answers: development experience and the introduction of atoms of confusion in the code snippets. The regression coefficient of not having an AOC was 1.27, while the coefficient of *experience* was 0.37. Subject's degree didn't have any significant value to impact the correctness.

### 3.3.5 Answer to RQ.1

Recalling from Section 3.1, our first research question is:

(RQ.1) Do code snippets that contain atoms of confusion produce a higher error rate than snippets where the atom is removed?

As the data in Table 3.1 shows there was only one atom whose correction has a small improvement in the percentage of correct answers. Varying from 7% to 132% improvements, nine out of the ten atoms, when present, made the code harder to understand. The logistic regression also showed that codes with AOC had a strong relation with wrong answers, indicating that the answer for RQ.1 is positive.

While development experience also had a impact on subjects' correctness, it isn't the main goal of this research to analyse this correlation.

### 3.3.6 Time Taken To Answer Questions

Table 3.2 displays the results of the measurements of the average time taken to answer with and without atoms of confusion. For this measurement, we have opted to exclude wrong answers from our analysis, for taking more (or even less) time to incorrectly predict the output is not relevant for our purposes. Once again, the Comma Operator was the construct whose removal had the greatest impact. The more expressive results in this measurement are the ones regarding Logic as Control Flow, Implicit Predicate and Omitted Curly Braces, as they are frequent in JavaScript libraries.[2]

Figure 3.5 shows a box plot of the difference in time taken the predict the output of correct answers. Although the dispersion is smaller for non-confusing code blocks, the median time did not vary much.

[2]Although our detection tool could not measure occurrences of the Logic as Control Flow atom, they were abundant in one of the repositories we used to perform the work described in Chapter 5.

Table 3.2: Difference in time taken(in seconds) to evaluate the output

| **Atom** | **AVG time** | **AVG time** | |
|---|---|---|---|
| | With AOC | Without AOC | $\Delta Time$ |
| Comma Operator | 60 | 21 | -0.65 |
| Logic Control Flow | 85 | 49 | -0.42 |
| Automatic Semicolon Insertion | 34 | 24 | -0.29 |
| Implicit Predicate | 33 | 24 | -0.27 |
| Omitted Curly Braces | 43 | 31 | -0.27 |
| Assignment as value | 53 | 49 | -0.07 |
| Arithmetic as Logic | 29 | 36 | 0.24 |
| Ternary Operator | 42 | 42 | 0 |
| Post Increment | 27 | 27 | 0 |
| Pre-Increment | 34 | 49 | 0.44 |

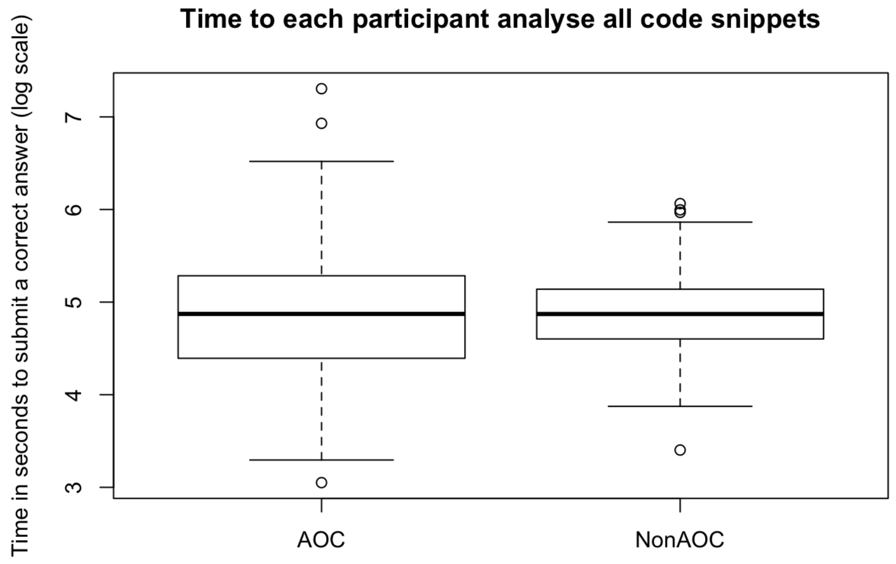

Figure 3.5: Box plot of time to answer each snippet

We also did a linear regression to see if there was any correlation between the time and other variables, like experience, degree, code snippet, and the introduction of the AOC. We measured just the correct answers and only the introduction of AOC showed a significant coefficient of 14, the rest of them didn't show any significant correlation.

## 3.3.7 Answer to RQ.2

Recalling our second research question:

(RQ.2) Do code snippets with atoms of confusion require programmers to take longer to predict their output?

As suggested by Table 3.2 and Figure 3.5, the variance in the time to correctly predict the output of a snippet is significantly smaller. Moreover, since the number of answers is the same for each inspected atom, data from Table 3.2 shows that the time taken to answer questions without atoms of confusion is 20% smaller. Not all atoms, though, took less time to predict the answer. In fact, for the Arithmetic as Logic and the Pre-Increment atoms, there was an increase of 24% and 44% in the time taken to answer questions wherein the atoms had been removed. For other two atoms, there no difference whatsoever in the time taken. Only in half of our atoms was there an increase in the time taken to answer. However, the linear regression did show some correlation of the time taken to answer correctly and the introduction of the atom. We suspect that some snippets were inherently harder to solve, even without the atoms, like the post and pre-increment, possibly affecting the linear regression.

We can then confirm RQ.2. While it didn't have a strong relation as subjects' correctness, the time taken to answers correctly overall did increase as the linear regression shows and Table 3.2.

## 3.4 Potential Downfalls Of Our Methods

Since our research was conducted online with unknown participants, we had no way of confirming their levels of education and experience. Also, as we briefly discussed in Section 4.3, we did not have a way to prevent respondents from cheating, such as running the code on an interpreter, or consulting other people. This presents a threat to the validity of the histograms presented in Figures 3.2 and 3.2.

Another threat to our results lies in the fact that some constructs might not be frequent, or present at all, in JavaScript code bases. The Automatic Semicolon Insertion and the Comma Operator atoms were not found in the code samples we consulted when formulating the questions, and we had to create blocks of code containing the atoms. The work done in Chapter 5 could serve as a starting point for detecting these atoms in a large data set of JavaScript files.

## 3.5 Conclusion

Regarding hypotheses H.2 and research question RQ.2, a potential problem with our method is that there was no strong incentive for subjects to think thoroughly about the questions. As we described in Section 3.4.2, we observed a lack of engagement when we ran the survey with undergraduate students. Although our final subjects were voluntarily partaking in the survey, we could not be sure that, after some time taking the survey, respondents would become tired and stop thinking clearly about the code. As questions were assigned randomly, and contained different levels of difficulty, comparing a subject's answer with their other questions would not provide a reliable metric.

In one of the atoms, namely the Omitted Curly Braces, we intentionally removed indentation from the code, which is highly unusual, given that many programmers use automatic formatting in their code editors. This can introduce some level of artificiality to this atom's question. On the other hand, while describing JavaScript's grammar to develop the parser we used for the work in Chapter 5, we discovered that the language allows the programmer to omit the curly braces after *if* statements, and insert multiple statements in the following line. This fact itself might constitute a source of confusion, which we leave to analyse in our future endeavours.

# Chapter 4

# Automatic Detection of Atoms of Confusion

This chapter describes work we have done in the development of a tool that automatically detects the atoms of confusion. Although the tool is still incomplete, and works under several assumptions and constraints, we are able to measure the occurrence of seven of the atoms. Due to these constraints, we were forced to be conservative when counting the atoms' frequencies. This means that the code bases we analysed contained *at least* as many occurrences as we detected, and probably more.

## 4.1 Detection Strategy

Once we have measured the impact of each atom of confusion, we are ready to proceed with automating the process of detecting them in production software, having the final goal of proposing their removal from source code, without affecting the program's behaviour. To do this, we apply metaprogramming techniques.

As defined in [15], a metaprogramming system is a programming system or language whose basic data objects include the programs and program fragments of some particular programming language. Such programs allow us take programs as input, and operate on them, potentially generating a modified program as output.

Metaprogramming allow us to streamline the process of detecting and removing atoms of confusion from large code bases. In this chapter, we describe a method for employing a metaprogramming language to automatically detect some of the atoms of confusion whose impact we have measured.

### 4.1.1 Metaprogramming Language of Choice

Rascal MPL is a standalone metaprogramming language, whose primitives include pattern matching, context-free grammars and concrete syntax for objects, amongst others. Although it does not compile or interpret code, Rascal MPL centralizes the tasks of parser generation and source code analysis/transformation. Generally, the stages depicted in figure 2.2 have to be implemented using more than one language. In such scenario, not only does the developer have to create tools for performing each of the steps, but they also need to dedicate time and effort into integrating all the parts. Rascal allows us to abstract this process away: by using it, we can describe a language's grammar, and the parser will automatically be generated for us. It also generates and provides visualization for abstract syntax trees, or ASTs, which are a type of intermediate representation that is needed for the scanning stage. Finally, it provides functionality to traverse and transform the tree, thereby allowing us to use a single tool to parse through code bases looking for atoms of confusion before altering the ASTs so as to automatically produce less confusing code.

One of Rascal's disadvantages is that it is still in its alpha versions, so its features are susceptible to frequent changes. Also, its documentation is not very clear, which makes users have to figure the problems out by themselves. Such disadvantages were minor compared to all the convenience and power the language provided, which led us to adopt it.

Once we devised an ES6 grammar and input it to Rascal, we used its Parser library to automatically generate the parser. We were then in a position to start traversing the abstract syntax tree. We proceeded to mine repositories containing JavaScript source code.

### 4.1.2 Subject Systems

We ran a Python script that mined the GitHub platform looking for projects whose core was written in JavaScript. After obtaining this list, we decided to choose subject-related repositories, and ended up selecting five projects used for developing user interfaces for web applications. Due to inconsistencies with the parser we generated for JavaScript, we had to perform simple manual changes to the projects source code, which prevented us for mining a larger number of repositories. These changes are listed in Section 4.4. The projects we mined were:

- Bootstrap: A library to build responsive, mobile-focused applications;
- Font-awesome: A popular vector icon set and toolkit

- jQuery: A well-established library, used in several web applications and frameworks, to introduce dynamism into web pages;
- Materialize: A library built upon Google's Material Design language, whose goal is to allow multi-platform unified user experience;
- SemanticUI: A user interface framework which uses natural language properties to create themes.

In the following section, we describe the assumptions and constraints imposed into the detection of each atoms. As we will explain at the end of this chapter, technical hindrances regarding the development of a fully representative ES6 grammar forced us into imposing such conservative conditions.

### 4.1.3 Criteria For Detecting Each Atom

For each of the atoms defined in Chapter 3, we formulated a set of conditions that must be met for our tool to automatically detect them in the code bases we analysed. The following list describes all the criteria that must be met for each atom to be detected.

1. Arithmetic as Logic: Any *if* statement whose conditional expression involved the operators $+, -, ++, --, *, /$ **and** did not contain any logical operators $(\&\&, ||, !=, ==, ===)$;
2. Assignment as Value: Any line that contained 2, 3 or 4 assignment operators;
3. Automatic Semicolon Insertion: Due to JavaScript's flexibility in allowing line breaks between many tokens, our current approach was not able to automatically detect the presence of this atom. We had to use regular expressions to detect the presence of *return* statements followed by a line break, and a literal in the following line;
4. Comma Operator: Any sequence of expressions enclosed by parentheses.[1]
5. Ternary Operator: At the moment of this writing, our tool was not able to automatically detect this atom. Therefore, we had to impose the following constraint: any expression where there is a statement followed by a colon, followed by another statement. This pattern was searched for using regular expressions.[2]

[1]During this stage of atom detection, we discovered that JavaScript allows another kind of statement sequencing. When separated by commas, and without parentheses, the statements will be executed from left to right. This construct can be used to pack multiple statements into a line where one would expect only one statement. We elaborate on this discovery in Chapter 6.

[2]This happened because we our parser was detecting ambiguity when searching for ternary operators. Also, we discovered that JavaScript allows for chaining of these operators. This is a potential great source of confusion, and will be left as a future research topic, as described in Chapter 6.

6. Implicit Predicate: The detection and evaluation of this atom would required semantic analysis, which is beyond the scope of this work. This is also left as a future endeavour.

7. Logic as Control Flow: The same problem as the previous atom applies. Since semantic analysis was necessary, we did not count the occurrence of these atoms.

8. Omitted Curly Braces: Any *if* statement that was not immediately followed by opening brackets, and that was followed by any of the following:

   - Expression
   - Iteration
   - *continue* statement
   - *break* statement
   - *return* statement
   - *throw* statement
   - *try* statement

9. Post Increment/Decrement: Any *id* token followed by ++ or −−

10. Pre Increment/Decrement: Any *id* token preceded by ++ or −−

## 4.2 Results and Analysis

Table 4.1 summarizes the results we obtained from the five projects we examined. As discussed in the previous section, results marked with an X correspond to atoms whose occurrence we could not measure without semantic analysis.

Although this sample is small, the projects were relatively large, since all contained between 20000 and 60000 lines of code. The absence of the Automatic Semicolon Insertion atom is, in hindsight, not surprising. Upon investigating the atom, we realised that, even though it is syntactically valid, its outcome is easy to catch. Upon returning an undefined value from a function call, the calling routine would try to operate on the value that has been returned. Since the undefined type cannot be operated in JavaScript besides in Boolean evaluations, the program would terminate when the calling function tries to manipulate the returned value.

It was also remarkable that three projects contained the Comma Operator. We were expecting it to be the most difficult to find, due to its eccentricity and difficult to understand. That fact that we could find the most confusing atom in three of a small sample of projects might suggest the atom is more frequent than we hypothesised. As we improve our grammar and detection tool, we intend to run it on a larger sample of projects, so as to have more data to analyse.

The Ternary Operator was the most abundant one. For all projects, it was the most frequent atom, which suggests that JavaScript developers are adept at using it. During this counting phase, we discovered that JavaScript allows the chaining of the Ternary Operator. When three or four of them were chained, the code looks rather confusing, and it became difficult to evaluate the final value returned by the expression. We leave as future work the task of applying a similar methodology of our survey in order to see if chaining multiple Ternary Operators increases the level of confusion.

Omitted curly braces appeared in all projects. The project where it had least occurrences was in jQuery. In fact, in the three times it appeared in that library, it was actually because it was present in a submodule, not in the library itself. Since jQuery was the second largest code base we inspected (about 42000 lines of code), this suggests that the developers might follow a style guide which warns against omission of curly braces. We also made the interesting discover that, when omitting curly braces, JavaScript allows many statements to still be executed inside an 'imaginary block' created by *if*, *while* and for statements. This means that the following code would perform all the statements in the second line:

```
if (age >= 21)
   canDrive = true, driveCar(), drinkWater()
```

Usually, when braces are omitted in C/C++ or Java, the following line contains a single statement. A semicolon would mean that only the subsequent statement will be contingent on the conditional. In these languages, separating statements with comma without parenthesising is not allowed, so this could could not work. This can also be subject of further investigation, since this is rare in other languages.

The post increment atom, our third most confusing atom, was also frequent in all projects, and abundant in jQuery. This would be our first candidate for automatic transformation, since removing this atom would not be difficult to do with Rascal MPL.

The least confusing atoms, namely the Pre-Increment and Arithmetic as Logic, were only more frequent than the Comma Operator and Automatic Semicolon Insertion. Since they were not regarded as very confusing by our survey's subjects, we would not focus on them as we developed the automatic transformation tool.

Table 4.1: of occurrences of each atom within each project

| Project | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Bootstrap | 0 | 0 | 0 | 1 | 426 | X | X | 49 | 109 | 4 |
| Font-awesome | 1 | 0 | 0 | 0 | 81 | X | X | 60 | 16 | 1 |
| jQuery | 9 | 141 | 0 | 0 | 592 | X | X | 3 | 526 | 53 |
| Materialize | 10 | 26 | 0 | 9 | 334 | X | X | 55 | 78 | 13 |
| SemanticUI | 25 | 13 | 0 | 30 | 143 | X | X | 18 | 60 | 0 |

## 4.3 Potential Downfalls of Our Approach

Since the beginning of this work, we have been working towards generating a parser for ES6. As we could not find any examples of a ready-to-use ES6 grammar, we had to start writing one ourselves. Our main references were:

- An ES5 grammar in Rascal's GitHub repository;
- The official ECMAscript specification.

However, this process revealed itself to be far more complicate than we expected. Not only did the ES6 specification contain significant changes to the language, but we also had the opportunity to see a vivid example of the trade-off between syntactical flexibility and grammar complexity. We knew from previous experience that JavaScript had a very flexible syntax, allowing line breaks between many of its tokens, 'on-the-fly' object declaration, omission of semicolon, omission of parenthesis for anonymous functions with a single parameter, and many other features. What we did not know was that this would make it dramatically harder for us to describe the language's grammar. Even with extensive help of our advisor professor, who had previously worked in describing a Java grammar for automatic refactoring with some of his master students, we were not able to address all the ambiguities that our parser indicated. All these problems were compounded even further by the fact that ECMAScript has released three newer versions of the specifications. Even though all of them are backward compatible, many projects adopted some of the newer features to develop their libraries[3]. As our parser was focused on the ES6 specification, it could not parse blocks of code that contained more recent syntax.

To work around this issues and present preliminary results, we applied the following changes manually to the code bases we inspected:

1. Removal of multiple variable declarations separated by line breaks: We collapsed all variable declarations into a single line;

[3]Amongst the projects we analysed, Bootstrap was the only to contain features of ECMAscript versions that were newer than ES6.

2. Removal of line breaks and comments between multiple variable declarations;
3. Removal of line breaks inside the conditional part of an *if* statement;
4. Removal of line break inside objects returned by *return* statements;
5. Addition of a new line at the end of every block enclosed by curly braces;
6. Commented the *spread* operator[4] occurrences, whenever they were present;
7. Removal of strings formatted with backslashes;
8. Removal of line break between the name of an object's property and its value (after the colon)
9. Tolerance for ambiguity during the parsing stage. This was necessary because any Ternary Operator occurrence that contained elaborate expressions on their right-hand side caused ambiguity.[5]

## 4.4 Conclusion

We can see that a lot of manual work had to be carried out in order for our parser to scan all the files in the projects we analysed. The most serious problem was allowing the parser to keep parsing even when ambiguities were detected. Although its damage to our counting procedures was that it caused us to detect the Ternary Conditional less times than they were actually present in the code bases, ambiguities in the generation of the abstract syntax tree are unacceptable if we are to propose automatic transformations. We leave as future work the task to improve and verify our parser in order to perform code transformations.

[4]The spread operator is used when we want to create a new object, with all the properties of another object, plus some more.

[5]This is the main reason why we opted to use regular expressions to create a conservative estimate of the occurrences of the Ternary Operator.

# Chapter 5

# Conclusions

In this chapter, we discuss the main challenges we faced, as problems encountered in the final stages of the project, which prevented us from deploying a fully functional code transformation tool, which was our initial objective.

## 5.1 Atom Impact

Regarding the results obtained from analysing answers collected during the survey we described in Chapter 3, we were satisfied with how the data confirmed our hypotheses that even experienced programmers can struggle to understand small snippets that contain atoms of confusion, and that these should therefore be avoided whenever possible. The results about the time taken to answer the questions were inconclusive, and we were not able to confirm the second hypothesis we had formulated. We think that we would need a more controlled environment, with more incentive to predict the output of the code.

Our levels of differences (the $\delta$ in table 3.1 between predicted output and corrected output were also satisfactory. Comparing ours with Gopstein's [4], we see that the values for Post Increment, Omitted Curly Braces, and Implicit Predicate stayed within close range, whilst others, such as Assignment as Value, presented very different measurements. Our atom whose impact was almost non-existent, the Arithmetic as Logic, was also found to be of little impact, which led the authors to not regard it as an atom in the end.

We were surprised to see that a very eccentric atom, the Comma Operator, with which none of the authors have yet dealt with during their work experience, were present in three of the five code bases we analysed. We also discovered two potential sources of confusion that exist in JavaScript, but are not possible in C/C++ or Java. These are:

- In the line following a conditional or iteration statement that contained code that was not enclosed by curly braces (i.e., in instances of the Omitted Curly Braces atom), JavaScript allows multiple statements to be executed, provided they were separated by commas;

- Ternary Operators can be chained, leading to intricate conditional logic, resembling multiple *if-else-if* statements.

We believe these two atoms can cause significant level of confusion, and would be great candidates for future exploration of sources of confusion in JavaScript

## 5.2 Atom Detection

We were somewhat frustrated with this part of the work, as we could not achieve the development of a parser that correctly analysed the code. Both the authors of the text work with JavaScript on a daily basis, and were impressed by how unintelligible its syntax can be. We were left with the impression that the flexibility the language shows to its users are compensated by a grammar that is difficult to describe. The fact that we could not find any ES6 grammar that was ready to use was also remarkable. In hindsight, we speculate that this is because the ECMAscript standards only specify the grammar, leaving the work of implementing it to other organizations. We were also caught by surprise, when attempting to count the atoms, because we found constructs we did not even know were allowed by the language. One of the projects we analysed contained so many single-line multiple statements separated by commas and chained ternary operators that we were left wondering if they would be present in many other projects. Future work in this front could occur in several directions:

- Thorough testing and debugging of our grammar, so as to have a complete parser for ES6. This would enable us to accurately detect any grammatical substructures allowed by the language we may find interesting, and/or confusing. This would also allow us to automatically parse a plethora of repositories, and have data that is orders of magnitude larger than that which we collected in this work;

- A comparative analysis of grammar complexity across different languages. In this sort of work, we would attempt to implement a correct parser for languages we imagine to have simpler grammars than JavaScript[1]

- Analysis of the constructs we discovered JavaScript allows that we were not aware of, and were apparently rather confusing.

[1]We speculate Ruby and Lua to be solid candidates, but we have no empirical evidence. However, BNF grammars for both languages can be found on the Internet, which should theoretically make their parsers' generation easier.

Although we were not able to implement a fully functional parser, we give a lot of credit to the creators of Rascal MPL. The tool, albeit very rich and somewhat complex, was quick to assimilate for our purposes, and few lines of code were enough to automate the detection the atoms, and to count their frequency.

In hindsight, we realised that we might have gained invaluable insight from inspecting libraries in JavaScript. One of the main objections we faced when running our survey was the fact that our code snippets seemed artificial. By then, we did not know better, and proceeded to write our own versions of confusing and non-confusing code. This might have led us to introduce bias, by introducing unnecessary complexity into the snippets, therefore forcing respondents to make mistake. But after analysing only five JavaScript libraries, we could find examples of all the atoms we surveyed, except from the Automatic Semicolon Insertion. We could have extracted these snippets, simplified them to make them independent of the program's context, therefore creating less partial questions.

Code Snippets presented on the survey

Arithmetic as Logic

Obfuscated ID: 11 Obfuscated Answer: false

```
let array_1 = [1,2,3];
let array_2 = [2,3,2];
if(array_1.length - array_2.length){
   console.log(true);
}
else{
   console.log(false);
}
```

Transformed ID: 1 Transformed Answer: false

```
let array_1 = [1,2,3];
let array_2 = [2,3,2];
if(array_1.length != array_2.length){
   console.log(true);
}
else{
   console.log(false);
}
```

Assignment as value

Obfuscated ID: 12 Obfuscated Answer: 0

```
function resetSchedulerState () {
   let activatedChildren = {length: 10};
   let index = length = activatedChildren.
    length = 0;
   console.log(index);
}
resetSchedulerState();
```

Transformed ID: 2 Transformed Answer: 0

```
function resetSchedulerState () {
   let activatedChildren = {length: 10};
   activatedChildren.length = 0;
   let length = activatedChildren.length;
   let index = length;
   console.log(index);
}
resetSchedulerState();
```

| Automatic Semicolon Insertion | |
|---|---|
| Obfuscated ID: 13 Obfuscated Answer: false | Transformed ID: 3 Transformed Answer: true |
| <pre>function example(){<br>   return<br>       10<br>}<br>if(example() == 10){<br>   console.log(true);<br>}<br>else{<br>   console.log(false);<br>}</pre> | <pre>function example(){<br>   return 10;<br>}<br>if(example() == 10){<br>   console.log(true);<br>}<br>else{<br>   console.log(false);<br>}</pre> |
| **Comma Operator** | |
| Obfuscated ID: 14 Obfuscated Answer: 13 | Transformed ID: 4 Transformed Answer: 3 |
| <pre>let V1 = 5, V2 = 10;<br>V1 = (V2 = 1, 2);<br>console.log(V1+V2);</pre> | <pre>let V1 = 5, V2 = 10;<br>V2 = 1;<br>V1 = 2;<br>console.log(V1+V2);</pre> |

Ternary Operator

Obfuscated ID: 15 Obfuscated Answer: 10

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

Transformed ID: 5 Transformed Answer: 10

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

Implicit Predicate

Obfuscated ID: 16 Obfuscated Answer: false

```
let V1 = 10, V2 = 3;
if (!(V1 % V2)){
   console.log(true);
}
else{
   console.log(false);
}
```

Transformed ID: 6 Transformed Answer: false

```
let V1 = 1, V2 = 2;
if ((V2 - V1) == 0){
   console.log(true);
}
else{
   console.log(false);
}
```

Logic Control Flow

Obfuscated ID: 17 Obfuscated Answer: 7

```
let V1 = 3;
let V2 = 5;
let V3 = 0;
while (V1 != V2 && ++V1) {
   V3++;
}
console.log(V1 + V3);
```

Transformed ID: 7 Transformed Answer: 21

```
let V1 = 1;
let V2 = 11;
let V3 = 0;
while (V1 != V2) {
   V1++;
   V3++;
}
console.log(V1+V3);
```

Omitted Curly Braces & Indentation

Obfuscated ID: 18 Obfuscated Answer: 4

```
let V1 = 1, V2 = 2
if (V1 > V2)
V2 = 1
V1 = 2
console.log(V2+V1)
```

Transformed ID: 8 Transformed Answer: 4

```
let V1 = 1, V2 = 2;
if (V1 > V2) {
  V2 = 1;
}
V1 = 2;
console.log(V2+V1);
```

Post Increment

Obfuscated ID: 19 Obfuscated Answer: true

```
let V1 = 0;
if (V1++ == 0) {
   console.log(true);
}
else {
   console.log(false);
}
```

Transformed ID: 9 Transformed Answer: true

```
let V1 = 0;
if (V1 == 0) {
   console.log(true);
}
else {
   console.log(false);
}
V1 = V1 + 1;
```

Pre Increment

Obfuscated ID: 20 Obfuscated Answer: 2

```
var index = 1;
while (++index < 10) {
   console.log(index);
   break;
}
```

Transformed ID: 10 Transformed Answer: 1

```
var index = 0;
index = index + 1;
while (index < 10) {
   console.log(index);
   index = index + 1;
   break;
}
```

# References

[1] Parr, Terrence: *Language Implementation Patterns.* The Pragmatic Bookshelf, United States - DF, 2010, ISBN 978-1-934356-45-6. `link`. ix, 6, 8, 9

[2] Knuth, Donald E.: *Literate Programming.* Comput. J., 27(2):97–111, May 1984, ISSN 0010-4620. `http://dx.doi.org/10.1093/comjnl/27.2.97`. 1

[3] Fowler, Martin: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999, ISBN 0-201-48567-2. 1

[4] Gopstein, Dan, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K. C. Yeh, and Justin Cappos: *Understanding Misunderstandings in Source Code.* In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 129–139, New York, NY, USA, 2017. ACM, ISBN 978-1-4503-5105-8. `http://doi.acm.org/10.1145/3106237.3106264`. 1, 2, 3, 7, 38

[5] McConnell, Steve: *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004, ISBN 0735619670, 9780735619678. 2

[6] Tilley, S. R., S. Paul, and D. B. Smith: *Towards a framework for program understanding.* pages 19–28, March 1996. 5

[7] O'Hare, A. B. and E. W. Troan: *RE-Analyzer: From source code to structured analysis.* IBM Systems Journal, 33(1):110–130, 1994, ISSN 0018-8670. 5

[8] Lämmel, R.: *Software languages: Syntax, semantics, and metaprogramming.* Springer, 2018. Book's website: `http://www.softlang.org/book`. 6

[9] Gopstein, Dan, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos: *Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild.* pages 281–291, 2018. 6

[10] Sipser, Michael: *Introduction to the Theory of Computation.* Cengage Learning, 2012. 7

[11] Louden, Kenneth: *Compiler Construction: Principles and Practice.* TNS, 1997. 8, 9

[12] StackOverflow: *StackOverflow Survey*, 2019. `https://insights.stackoverflow.com/survey/2019`, visited on 2019-06-05. 12

[13] George E. P. Box, J. Stuart Hunter, William G. Hunter: *Statistics for Experimenters: Design, Innovation and Discovery*. Wiley-Interscience, 2004, ISBN 9780471718130. 20

[14] Kuehl, R.O.: *Design of Experiments: Statistical Principles of Research Design and Analysis*. Duxbury/Thomson Learning, 2000, ISBN 9780534368340. 20

[15] Cameron, Robert D. and M. Robert Ito: *Grammar-Based Definition of Metaprogramming Systems*. ACM Trans. Program. Lang. Syst., 6(1):20–54, January 1984, ISSN 0164-0925. `http://doi.acm.org/10.1145/357233.357235`. 31