Atom Candidates Considered in the Research

Double Blind

May 3, 2021

Abstract

This documents presents the set of JavaScript' Atom Candidates we considered in our research. Our choice of atoms was primarily based on syntactical constructs in C that are also valid in JavaScript, with a particular atom that was exclusive to JS. In this docment we briefly discuss each atom, and describe their potential for confusion. We also present an equivalent version of the code without the atom.

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.

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
```

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.

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
}
```

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
}
```

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')
}
```

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()
}
```

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()
```

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
```

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
```