@JavaScript: Augmented JavaScript

Iosif Iulian Petrila *

Posted Date: 3 February 2025

doi: 10.20944/preprints202502.0081.v1

Keywords: JavaScript; @JavaScript; augmented language; self-implementing language; bootstrap system; universal system language; ubiquitous computing instrument

 doi:10.20944/preprints202502.0081.v1

Article

@JavaScript: Augmented JavaScript

Iosif Iulian Petrila

Faculty of Automatic Control and Computer Engineering, Gheorghe Asachi Technical University of Iasi, Str. Dimitrie Mangeron, Nr. 27, 700050, Iasi, Romania

* Correspondence: iosifiulianpetrila@gmail.com

Abstract: The augmented JavaScript programming language completed as general computing instrument concept, referred as @JavaScript, is presented. The augmentation consists in the minimalist dialectal completion of the language, keeping compatibility with the standard version, in order to transform the language into a general-purpose language and to expand its use as universal programming instrument suitable in any computing system areas and for any type of computing device. The augmentation includes integrating existing high and low level elements along with introducing new elements in order to be transformed into a bootstrap-type self-hosting language, scalable and flexible in being usable in any type of computing processing and contexts as a ubiquitous instrument, suitable for incorporating natural languages and other present and future computing paradigms.

Keywords: JavaScript; @JavaScript; augmented language; self-implementing language; bootstrap system; universal system language; ubiquitous computing instrument

Introduction

Similar to the "divine interests" that mixed people's languages in the "Babel Tower", within computer systems, a similar mixture of languages has emerged, with languages dedicated only to a specific computing task or aspect, even if in essence one language does not differ much from another. The process of essentialization and focus on relevant languages will likely be accelerated by the fact that in computing it starting the use of natural language in programming to an increasingly greater extent, therefore, at least in the transition period, high-level languages must facilitate this approach by augmenting it with both higher-level elements related to natural language and lower-level elements from the area of system programming, an area currently dominated by mid-level typed languages with some elements of low-level languages. The most promising language in this endeavor, suitable also for universalisation, is practically JavaScript language, because it is also the most avant-garde high-level language (it introduces the most representative concepts such as inclusive type checks, syntactic asynchrony, promises, etc.) but especially because it is already integrated into computing systems as a standard within the web area, every minimally relevant current computing system (with a browser) having at least one interpreter of it, which makes it an indispensable language for a computing system, unlike any other language.

Over the time, the JavaScript language landscape itself was the subject of similar mixture of versions, variants, adaptations, directions and transformations, such as: Mocha (initial name), LiveScript (second name), JavaScript (Netscape name), JScript (Microsoft's variant), SpiderMonkey (Netscape & Mozilla engine), DOM (W3C Document Object Model for interaction with web documents), Rhino (server-side engine), JScript.NET (JScript adaptation to .NET static ecosystems), ECMAScript (standardized JavaScript versions), AJAX (Asynchronous JavaScript and XML data exchange features), jQuery (concise selector syntax and cross-browser compatibility library), V8 (Google engine), Node.js (server-side script adaptation), CoffeeScript (syntactic alternative), CommonJS (standardizing server-side modules), ClojureScript (language transpiled to JavaScript), Electron (cross-platform hybrid application framework), React (Facebook UI library), JSX (syntax

 doi:10.20944/preprints202502.0081.v1

extension for React), Babel (backward compatibility features), Webpack (module bundler and build tool), TypeScript (Microsoft enhancing static typed safety transpiled superset), Vue (progressive framework for building UIs), React Native (Facebook mobile apps framework with UI as HTML syntax-level embedding), Angular (enterprise-scale application framework), WebAssembly (binary instruction format), AssemblyScript (TypeScript variant for WebAssembly compilation), Deno (secure runtime for JavaScript and TypeScript), Bun (all-in-one runtime bundler), etc. [1–17]

However, despite its many evolutionary paths, with transformative directions reaching the peak during the Smartphone revolution, JavaScript language has always revolved around its basic utility in web development, with extensions that represented more horizontal completions of the language, and less developing vertically in order to become a self-compiling language and leveraging existing low-level (TypedArray, WebAssembly etc.) features along with some new one in a general programming instrument [18] which may partially intersect and complement in lower-level aspects other similar one [19] but also to integrate higher-level elements with higher-order abstractions related to nature-inspired computational methods, paradigms, and natural language integration [20].

@JavaScript

The exegesis of augmentation language concept meaning, in the present context, requires highlighted some considerations regarding to language levels, especially since are defined differently across the literature. A programming language facilitates the description of some operations that a computing system should perform in a given context. Since humans find it difficult to work directly with numerical codes (sequences of bits and bytes representing processor instructions as words related to a specific instruction set architecture) which represents computers natural machine language (the only language that a computing system can directly understand), we use descriptions that are more convenient for us, closer to our mathematical or even our natural language through programming languages with a higher level of abstraction than that of the machine, descriptions that must be converted into machine language (compiled) in order to be executed or interpreted and practically understood by the computer. Depending on the level of abstraction (distance from the machine or more generally to target/destination entity), there are four categories of languages: low-level languages (close to the machine level, machine-dependent such as machine language and assembly language), mid-level languages (minimal abstraction with informational elements similar to machine but in machine independent and portable way, these are usually typed system languages such as C, C++, etc.), high-level languages (which operate with abstract elements similar to mathematical elements and closer to our language, these being untyped languages such as JavaScript, Python, etc.) and natural-level languages (which use descriptions in natural languages). Obviously, these low or high categorizations can be viewed even more generally in relation to the proximity of the language to either the target or the source entity (for example, a parrot uses from his perspective a low-level language, closer to its target, when imitating us). A dialect is a language that is largely compatible with the standard one, but which has additional specific language elements to address particular needs, contexts or environments. Regarding the level languages structuring, two main categories of dialectal languages can be identified: extensions and augmentations. Extensions complete a language horizontally, maintaining its level (like is C++ in relation to C, which extends it but within the same mid-level). Augmentations complete a language vertically, adding both lower-level and higher-level elements (as is @C in relation to C). Therefore, from this perspective, an augmented language can include elements specific to each level (even if they will not be as efficient a level specific language), including facilities to operate with elements specific to machine languages but also with higher level such as natural languages. Thus, @JavaScript represents an augmented version of the JavaScript language, different from the extension type dialects of the language.

Over the time, the JavaScript language has undergone more to extension-type transformations, augmenting it with both lower-level and higher-level elements being a challenge because of its dynamic nature, which, although it facilitates the incorporation of higher-level elements, at the same time makes it difficult to manage low-level elements that prefer static structuring. The augmentation

 doi:10.20944/preprints202502.0081.v1

with lower (mid and low) level elements is intersected with the similar augmentation process but with higher-level elements in the case of the C language [19], thus creating a compatibility window between the two languages with the potential for mutual capitalization of existing facilities and resources, obviously this assumes augmentations with: metaprogramming capabilities, mixing static with dynamic codes, strong typing (beyond TypeScript, at least to separate integers from reals, etc.), flexible dynamic type encodings, multithreading (beyond Web Workers), support for SIMD operations, direct memory and system access facilities, etc. These low-level augmentations provide the foundation for direct hardware access and native code generation, enabling JavaScript to operate efficiently at the system level without intermediary layers, opening up new possibilities for embedded development and IoT applications, while maintaining the language's security model through contextual restrictions. The low-level facilities must ensure the criterion of minimal relevance, meaning they must allow the language to become self-compiling. The compilation features must complements existing WebAssembly solutions by providing specialized performance optimizations for computationally intensive tasks: while WASM excels in security-critical applications through its sandboxed execution environment, the augmented variant must combines traditional JIT (just-in-time) compilation for dynamic code with selective AOT (ahead-of-time) compilation paths for performance-critical sections, enabling optimal hardware utilization for demanding workloads such as: ML inference, GPU compute operations, real-time data processing, etc. This multi-faceted approach allows developers to leverage the security benefits of WASM, the dynamic flexibility of JIT, and the performance advantages of static compilation based on their specific requirements and contexts (web client or server, compiled application, shell tasks, etc). Some low-level facilities already introduced into the language, such as TypedArray elements, need to be made more flexible and completed in terms of internal operations with heterogeneous data, including with a single data type and not just with homogeneous blocks, but also in terms of their initialization at the user level. Also, in addition to the usual formats (JSON, XML, Binary WASM, etc.), WebAssembly specific codes need to be integrated into the language in both WAT (WebAssembly Text format) through S-expression descriptions

```
(func $add
  (param $a i32)
  (param $b i32)
  (result i32)
  local.get $a
  local.get $b
  i32.add
)
```

and also through an equivalent semi-parsed format based on common arrays, WAA (WebAssembly Array format)

```
['func', '$add',
  ['param', '$a', 'i32'],
  ['param', '$b', 'i32'],
  ['result', 'i32'],
  ['local.get', '$a'],
  ['local.get', '$b'],
  ['i32.add']
]
```

 doi:10.20944/preprints202502.0081.v1

that will be easier processed and serializable into JSON-like format. The augmented JavaScript achieves a minimalist flexibility by remaining compatible with the standard version while introducing enhanced parsing facilities for distinct code areas related to different related languages (such as HTML, CSS, etc.). Unlike drastic syntax modifications seen in frameworks like React Native (that effectively creating a new hybrid language), the augmented approach relies on recognizing delimiting landmarks compatible with both HTML and JavaScript as in Web client context. This allows the related languages to remain distinct, while improving compatibility and facilitating parsing of HTML client descriptions for diverse contexts, including compilation. The augmentation not only facilitates hybrid application implementations, as can be seen in some existing framework where the application is a browser with a non-restrictive access to system resources, but also supports modular compilations, optimized resource usage, and seamless integration of diverse programming components, similar to any other type of compiled language. Thus, the language become more flexible by tolerating mixed code areas that are managed depending on the processing context (web client or server, compiler, etc.), for example, JavaScript code can be delimited by the HTML `</script>` tag in other contexts as well and not only in a web client context

```
// JavaScript code here
</script> <!-- End JavaScript code -->
```

Also, some language methods must be made more flexible, becoming contextual, along with the introduction of new ones as in the following examples

```
// Server-side/context
document.write("Hello from Server!"); // Send to the client
echo("Hello from Server!");           // Also send to the client
```

or similar content but in a web client context

```
// Client-side/context
document.write("Client Message"); // Overwrite body document content after
load
echo("Client Message");             // Append to body document even after
loading
```

features that will allow coding uniformization regardless of the context used, eliminating the gap between server and client and also between web and system development, making information exchange more flexible with both asynchronous and synchronous operations, allowing seamless integration between frontend, backend, and native components while preserving the language's elegant simplicity and familiar paradigms.

JavaScript was initially designed with inherent flexibility and semantic resilience to serve web clients, aiming to handle all possible meanings with minimal errors. These foundational characteristics, deeply embedded in its architecture, make it uniquely suitable for natural language augmentation, where semantic interpretation and contextual understanding are paramount. Ironically, although JavaScript's semantic flexibility has been a source of jokes and memes (see unexpected implicit operation conversions that surprise beginners), this very characteristic mirrors natural languages themselves because natural languages also facilitate wordplay, linguistic jokes, and multiple interpretations of meaning. Just as human languages thrive on semantic ambiguity and contextual interpretation, JavaScript's flexibility becomes an asset rather than a liability in natural language processing contexts. However, augmentations with higher-level elements, such as with natural language elements, bring their advantages and limitations, especially due to the non-

 doi:10.20944/preprints202502.0081.v1

deterministic nature of natural language [20] and can be combined with more formal elements [18]. Most of high-level augmentations are closely related to some low-level elements, such as natural specific calculations that demand broadcasting operations, operators overloading and function vectorization (tensorization) in order to describe matrix (tensor) operations more concisely which require low-level implementations for efficiency. In order to maintain compatibility with the standard version of the language, the augmentations must be contextually active at the language level being rather more specific to the type of processing instrument context: interpreter (client, server, shell, etc.), compiler, translator, transpiler, etc.

Conclusions

The JavaScript augmentation encompasses both system-level capabilities and computing primitives, enabling the development of core computational instruments based on self-implementing language architecture supported by a bootstrap system reflected in intrinsic implementation and compiling facilities. However, @JavaScript concept assumes augmentations that are effectively more related to the JavaScript code processing instrument enhancements in order to manage codes in different contexts (interpreter web client or server, compiler, etc.) and less about the language itself, which only requires minimal flexibility and completions.

Although there are multiple variations of JavaScript, most of them targeting particular horizontally extended aspects, the current augmented JavaScript concept is mainly a scientific one related to computing languages and afferent optimal processing instruments, which requires a flexibilization endeavor, starting from the reality of the language integrated existence in any minimally relevant computing system connected to a network and highlighting a minimal upgrading according to the principle of minimal completion with elements that would transform it into a universal ubiquitous computing instrument with utility ranging from system to application programming and also suitable of incorporating other present and future computing paradigms.

References

1. B. Eich, C. R. McKinney, *JavaScript Language Specification*, Netscape Communications 2, 1996.
2. Microsoft, *JScript*, MSDN Documentation, 1996.
3. ECMA International, *ECMAScript Language Specification*, 1997, https://tc39.es/ecma262
4. World Wide Web Consortium, *Document Object Model*, 1998, https://www.w3.org/TR/WD-DOM
5. Microsoft, *JScript.NET*, MSDN Documentation, 2002.
6. Mozilla, *JavaScript*, MDN Web Docs, 2005, https://developer.mozilla.org/en-US/docs/Web/JavaScript
7. J. Resig, *jQuery Library*, 2006, https://jquery.com
8. Google, *V8 JavaScript Engine*, 2008, https://v8.dev
9. R. Dahl, *Node.js: Evented I/O for V8 JavaScript*, JSConf EU, 2009.
10. OpenJS, *Node.js*, 2009, https://nodejs.org
11. A. Gal, et al., *Trace-based just-in-time type specialization for dynamic languages*, ACM Sigplan Notices 44 (2009) 465-478.
12. S. Tilkov, S. Vinoski, *Node.js: Using JavaScript to build high-performance network programs*, IEEE Internet Computing 14 (2010) 80-83.
13. Facebook, *React: A JavaScript Library for Building User Interfaces*, 2013, https://react.dev
14. M. Haverbeke, *Eloquent javascript: A modern introduction to programming*, No Starch Press, 2018.
15. B. Cherny, *Programming TypeScript: making your JavaScript applications scale*, O'Reilly Media, 2019.
16. World Wide Web Consortium, *WebAssembly*, 2019, https://www.w3.org/TR/wasm-core-2
17. A. Wirfs-Brock, B. Eich. *JavaScript: the first 20 years*, Proceedings of the ACM on Programming Languages 4 (2020) 1-189.
18. I. I. Petrila, *Implementation of general formal translators*, arXiv:2212.08482 (2022).

 doi:10.20944/preprints202502.0081.v1

19. I. I. Petrila, *@C – augmented version of C programming language*, arXiv:2212.11245 (2022).
20. I. I. Petrila, *Neural Information Organizing and Processing – Neural Machines*, Preprints 2024031043 (2024).