The GPM meta-transcompiler: Harmonizing JavaScript-oriented Web Development with the upcoming ECMAScript 6 "Harmony" Specification

Kyriakos-Ioannis D. Kyriakou, Ioannis K. Chaniotis and Nikolaos D. Tselikas
Department of Informatics and Telecommunications, University of Peloponnese, End of Karaiskaki street, 22100 Tripoli, Greece

***Abstract*— In this paper we present the incompatibility and complexity problems in Web development when using JavaScript. We also demonstrate the solution given by transcompilers and/or languages that compile to JavaScript, but we also highlight the respective problems. Thus, we present our proposal, i.e., a meta-transcompiler, that targets to harmonize JavaScript-oriented web development and tight the feedback loop between web standards' editors and web developers, even during the transition period of work-in-progress specifications, such as the upcoming ECMAScript 6 "Harmony" specification.**

Keywords—JavaScript; ECMAScript; transpilers; compile-to-Javascript languages.

I. INTRODUCTION

JavaScript was developed in 10 days by Brendan Eich in 1995 at Netscape for the purpose of complimenting Java as a client-side scripting language and implementing a design to represent the next generation of software designed specifically for the Internet [1]. One of JavaScript's core ideas was to be an open and cross-platform language, to be used easily by people with little or no programming experience, to quickly construct complex applications, such as HTML page authors as well as application developers, and to dynamically script the behavior of objects running on either the client or the server [2].

ECMAScript is the name of the official standard, with JavaScript being the most well known of the implementations [1]. Currently the implementations of the language are based upon the ECMAScript 5 (ES5), and the upcoming version, expected to be released, is ECMAScript 6 (also known as ES6 "Harmony").

According to the TIOBE Programming Community index, JavaScript is among the 10 most popular programming languages, and its usage has risen since last year [3]. The Web has become an important platform for application development and JavaScript is the only language to be found in all major browsers. Its association with the browser is one important reason that makes it one of the most popular programming languages in the world. This fact has given birth to a number of languages that compile to JavaScript, in order to appeal to programmers not comfortable with the language [4].

New developments, such as Node.js, allow also the usage of JavaScript on the server-side [5]. Thus, the potential of client-server programming integration for the web has emerged, by utilizing the same language end-to-end. One of the key factors that attributed to the rapid growth of JavaScript, is also the module ecosystem of Node.js, which is built around the Node Packaged Modules (NPM) service, because of its open-source nature and robust packaging systems, with registry [6]. As of April 2013, JavaScript and Node.js had already surpassed any other language/platform in number of packages contributed per year [7]. According to Module Counts, a service that gathers daily statistics based on the contribution of modules retrieved from popular package managers, JavaScript appears to currently have the largest ecosystem with 103044 published modules, followed by Java with 85592 modules (as measured on the 4th of September 2014). Furthermore the average daily growth of contributions of JavaScript code measures 256 daily published modules, more than four times the average number of daily published modules of Java code [8].

JavaScript is also responsible for offering cross device coverage. What all platforms have in common, is the increasing compliance to the web standards brought to them through modern web browsers. Technological achievements, such as the improvement of web browsers and the standardization of HTML5, led to the creation of advanced web applications, offering features, such as real-time communications, data on geographical location and device specific hardware such as the camera, the accelerometer etc, only available to native applications before. We have successfully produced such an application that was demonstrated live, as a proof of concept [9]. The findings and results of our previous work indicate that JavaScript is highly recommended for building full-stack web applications, addressing the growing demands in concurrency as well as the productivity, because of the overall gains observed. Our final verdict towards the popular question among programmers, whether JavaScript is a viable option for building modern web applications or not, is a definite "yes" [10]. However, JavaScript doesn't come without any drawbacks.

The rest of the paper is organized as follows: In Section II, we state the incompatibility and complexity problems of using JavaScript in web development. Section III describes the solution given by transcompilers and/or languages that compile to JavaScript, but also highlights the respective problems. In Section IV we present our solution, i.e., a meta-transcompiler, to all the afore-mentioned problems. The paper concludes and presents our future plans in Section V.

978-1-4799-6390-4/15/$31.00 ©2015 IEEE

II. JAVASCRIPT IN WEB DEVELOPMENT: THE CHAOS

Despite its great flexibility, JavaScript falls short on various native design aspects. E.g., its automatic type coercion can lead to unexpected behavior if strict equality operator (i.e., "===") is not used. Floating point numbers lack precision. Automatic semicolon insertion can break return statements. Function hoisting combined with function scoping can lead to unexpected behavior. The ability to declare a global variable by omitting the `var` statement and lack of tail call optimization are just a few of the peculiarities lying behind the programming language of the web [11]. Of course, this is just a short list of JavaScript's native aspects, but all these are already known to web developers or can be found in any JavaScript textbook. In the rest of the Section, some trickier aspects of JavaScript, i.e. incompatibility and complexity issues, are following.

A. Incompatibility Issues and Standards' Deviations

In theory, a standard is specified to provide compatibility between different implementations. But in practice, this is extremely difficult to be achieved 100%, since, usually, the majority of implementations deviates from the standard. This happens in the case of ECMAScript standard, too. In theory, browsers and JavaScript's engines have to follow the current ES5 standard, but in practice we still meet slight or, in some cases, substantial ES5 deviations and incompatibilities [12].

Furthermore, the upcoming version, i.e., ES6 Harmony, is currently a work-in-progress and promises various bugs and native aspects' fixes of the previous versions. At the same time, ES6 will mark a new era for JavaScript as the releases will become a lot more frequent [13]. Although ES6 will offer updates and fixes when released and implemented by browser vendors, older engines that won't be updated will not be compatible with applications making use of any new language features. More frequent announced changes to the specifications will only narrow down the margin of inconsistencies, since JavaScript implementations by browser vendors do not strictly follow them, as pre-mentioned. And this is not the only problem. Even if browser vendors do not fully conform to the existing specs (i.e., ES5), they tend to follow the current working draft (i.e., ES6) by partially implementing and provide it to developers, in order to get acquainted with the changes and upcoming features [13]. To aid the transition to ES6, various tools, such as polyfills and transcompilers are used, which are thoroughly described in the next section [14-17]. These tools have emerged to help abstract away the complexity, especially in front-end development, where the composition of different platforms with multiple devices and browsers creates a highly fragile environment for client-side scripting.

B. Complexity Issues: JavaScript's Complementaries

From its concept, JavaScript wasn't implemented to be a full fledged language to build applications, but rather to accommodate Java in client-side scripting. But, because of its recent rapid expansion, workflows and tooling have emerged, mainly in the form of open source projects, to complement the language and provide the useful features that other languages and integrated development environments (IDEs) offer. On the other hand, Javascipt's complementaries add also extra complexity to web development, as addressed below.

1) Module Loaders

Traditionally in order to load JavaScript in a browser, a script tag (i.e., `<script>`) had to be appended in the HTML, preferably at the end of the page to avoid extra rendering delay. As the number of scripts in a page increases, it becomes difficult to keep track of the dependencies, since the only clue is the sequence of those scripts. Moreover, updating the dependencies means that each script has to be changed manually. Conflicts in the global scope are likely to happen if a library won't check the `.window` properties before initialization.

CommonJS (CJS), is the core module loader for Node.js [18]. In CJS each JavaScript file is treated as a module in its own sandboxed scope. Functionality can be exported or imported using the `module.exports` and `require` properties, respectively [18]. Although CJS works out of the box in Node.js, it requires a client library such as SystemJS or offline bundling of the scripts to work on the browser [19-20]. The Asynchronous Module Definition (AMD) API specifies a mechanism for defining modules such that the module and its dependencies can be asynchronously loaded [21]. ES6 modules are currently not standardized nor implemented in any major browser [12]. Thus, to maintain consistency in a project by supporting all the aforementioned specs a universal module loader or a build process is a necessity.

2) Package Managers

NPM is a package manager for Node.js. Packages can be installed or published by typing the corresponding commands (i.e., `npm install` and `npm publish`, respectively [6]). Package dependencies are handled automatically, while many useful analytics, such as the most depended upon and downloads per day are available in the NPM registry.

NPM initially hosted only server-side packages. Front-end packages were eventually added, e.g., jQuery [22]. Up to date more package managers have been introduced to the JavaScript ecosystem, i.e., Bower, Jam, jspm and duojs focusing solely in front-end libraries [23-26]. To move in production, a package manager can be integrated with a build step that concatenates and minifies all the dependencies.

3) Sharing DOM

The Document Object Model (DOM), even if it is often accessed by using JavaScript, it is not part of it and it can be used by other languages too [27]. That is to say the DOM is loosely coupled with an application's state and it has to be updated manually to represent it. Moreover, when JavaScript accesses DOM, the rendering associated computations happen on the main thread of the browser. Thus, performing heavy DOM's manipulation can deteriorate rendering performance.

To overcome this problem, React, a JavaScript library for building user interfaces, uses a virtual DOM diff/patch implementation [28]. The development is optionally done using JSX, a JavaScript embedded markup language that transforms into optimized native JavaScript [29].

978-1-4799-6390-4/15/$31.00 ©2015 IEEE

4) Task Runners

During development, or when moving into production a build process is a viable solution to avoid repetitive tasks. Other than the OS based tools, various task runners have emerged, such as Grunt, Gulp, Broccoli and Cake [30-33]. NPM, mentioned above, also provides a simple mechanism to perform such tasks. Task runners provide basic functionality out of the box, such as controlling the flow of tasks and watching files for changes.

III. Transpile to JavaScript: A Solution or a Problem?

An alternative attempt to fix quirks and inconsistencies of JavaScript, as well as extend its features when used in the production of complex real world applications, is to use a transcompiler. A transcompiler (also called source-to-source compiler or transpiler) is a type of compiler that takes as input the source code of a programming language and outputs the source code into another programming language. Currently, a large number (more than 230) of transcompilers has emerged [34]. Some are made to appeal to developers already accustomed to a specific workflow, like TypeScript by Microsoft, incorporated in Visual Studio [35] and Dart from Google, which plugs in Eclipse [36], while others are more generic like CoffeeScript by Ashkenas [37].

Based on the above, converting JavaScript source code written in ES6 to source code that follows ES5 standard is also a transcompilation process that aims at ensuring compatibility. In order to further investigate the interoperability of the tools presented in the previous section, we implemented a web application, backed by an automated build process that transpiles ES6 code snippets to ES5 and executes them on the browser [38], as depicted in Fig.1. We examined this case, because it will be part of every build process after ES6 standard finalization, in order to support previous engines.

Traceur, i.e., an ES6 to ES5 transcompiler by Google was used as the core of the build process [39]. Its noteworthy restraint is that, even the code is compiled on the server, for some supported features e.g., classes, destructuring and array comprehensions, a runtime is also required to be bundled with the code [39]. Despite that Traceur comes with many ES6 features, modules and proxies were not fully supported, thus, SystemJS and Harmony-Reflect had to be used to provide the corresponding functionalities, respectively [19], [14]. Version mismatches between any of the aforementioned runtimes and Traceur lead to unexpected behavior and cryptic errors. In order to workaround this issue and keep the versions synced we used NPM, since it provides packages for both the client and the server. After we managed to find a working combination of versions, `npm-shrinkwrap` was proven very helpful, since it froze the versions of all packages [40]. Gulp was used to coordinate the build process and Express to host the application [31], [41].

At this point we had established a workflow, albeit rather complex and restrictive. Extending Traceur with features implemented in other compile-to-JavaScript languages, i.e., Coffescripts' classes and destructuring that do not require an additional runtime, is out of the equation. The transcompilers can not be chained, since their input is not compatible with the previous output. We also performed the Regenerator [42], a transcompiler by Facebook engineers, in a Fibonacci number calculation and we observed a 162.6 times slower execution time from the original source [38].

The fact that ES6 will allow developers to use features of the language that the majority of them was inspired and implemented by CoffeeScript since 2009, which unlike ES6 are backwards compatible without the usage of overhead inducing polyfills, is a strong indicator of the need of such languages.

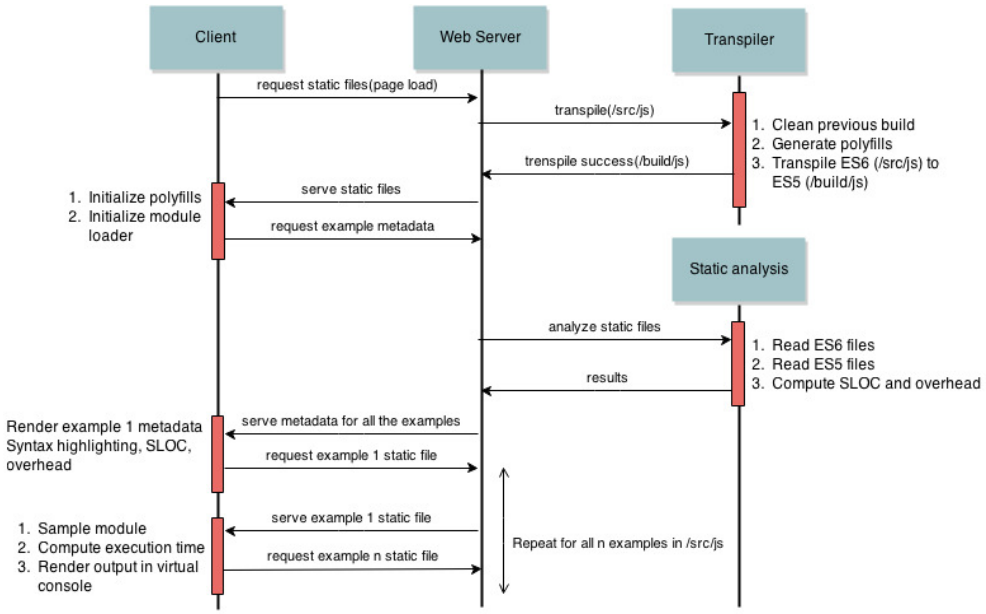

Fig. 1. Transcompilation (ES6 code to ES5) and execution process

Another reason CoffeeScript is useful, as well as other languages that compile to JavaScript, is that it can help developers make smaller and more readable, thus easier to maintain, programs by simplifying the JavaScript syntax and making each line easier to comprehend [43].

Still, relying on monolithic transcompilers is not a cure by itself. For example, newer advancements like generator functions, included in the ES6 specifications haven't made it to any of the known languages that output JavaScript, e.g., CoffeeScript, although they have been used in server side programming for over a year now and specific tools have been built around this new functionality, e.g., koa for writing "expressive middleware" and co, adding improved asynchronous flow-control as well as error handling [44], [45]. Publishing open source programs written in a relatively novel, obscure or even unknown language defeats part of the whole purpose of collaborating with others as well. Although languages that output JavaScript address JavaScript issues, they don't offer interoperability among them. Furthermore the output of a transcompiler is not always readable JavaScript code, leading to dependence on the chosen language, which is not standardized and may come with a steep learning curve as well as quirks of its own, only to be discontinued at some point in the future.

To conclude, languages that compile to JavaScript and monolithic transcompilers, as they stand today, although they have served as insight for the ECMAScript, don't come without shortcomings of their own, hence a better solution is required.

IV. The GPM meta-transcompiler

The Extensible Web Manifesto, signed by many TC39 (ECMAScript) and W3C members and others, declares the notion of many web related individuals, that "want to change how web standards committees create and prioritize new features" and "aim to tighten the feedback loop between the editors of web standards and web developers" [46].

Our main proposal moves in parallel with that cause. We base our idea on the great contribution of the NPM service to the JavaScript open source ecosystem, as well as the success of languages that compile to JavaScript have at extending the language and fixing various inconsistencies. In order to harness the power of JavaScript today, while being aware of the compatibility levels that can be achieved and at the same time offering the syntax and features that are best oriented towards a specific coding style and demands, we propose a new way of composing JavaScript code, to accommodate any programming environment while separating and abstracting most concerns encountered by modern web application developers.

We propose Grammar Packaged Modules (GPM), i.e., a meta-transcompiler that uses packages similar to NPM, containing syntactical transformations that can be any set of rules, dictating the strict transformation of any valid input. For proof of concept we used sweet.js, a library by Mozilla adding Lisp style macros to JavaScript [47], [48].

Unlike other existing transcompilers that come full featured by default, GPM meta-transcompiler is not able to perform any actions. GPM is simple in concept while makes no compromises in terms of flexibility and power for the package developer and at the same time abstracts low level implementation details from the end developer. It is tightly coupled with open source concepts, as well as the current and future ECMA specifications, assuring future maintainability and contribution.

The only required component to set up a workflow is `project.json` (Fig. 2). Further actions can be performed according to the fields specified in this file. Since GPM has no functionality on its own, in order to successfully transpile a project's source files the fields 'grammarDependencies' and 'targetSpec' that describe the syntactical transformations and the output ECMAScript specs (e.g., ES5) respectively, are required.

Published GPM packages can be installed with `gpm install`. This command installs a package and any packages that it depends on (Fig. 2). The dependency tree is built by satisfying all sub-dependencies up to the deepest branch while avoiding duplicates that may have already been fetched at a higher level, as with the `npm install` command [6]. Projects can be published as self contained packages of grammatical transformations, as with the `npm publish` command [6]. All published packages can be installed by name in other projects.

To enumerate, *project.json* can be extended with the following fields:

- *.name* and *.version*, are used to set the package name and semantic version respectively
- *.repository* contains the type and url of the version control
- *.documentation* contains a path or a url of the documentation
- *.reserved* points to a local file or directory that contains GPM specific configuration, this can be a mapping of the reserved words used for syntactic transformations with their functionality, similar to the structured mappings of esprima [49]. That provides optional but rather useful features like adding support for editors and IDEs via syntax highlighting, linting and automatic conflict resolution, i.e., if more than one modules use the same reserved word
- *.test* specifies the folder where the tests to be run are located

In order to further enhance the development process, GPM should provide extendable infrastructure for code linting, syntax highlighting, source mapping and automated compatibility testing. Because the language is dynamically generated, operators and keywords can not be known beforehand, thus the rules for the linter have to be generated in parallel with the grammar, i.e., after all the dependencies have been resolved. The same principle applies to syntax highlighting and documentation.

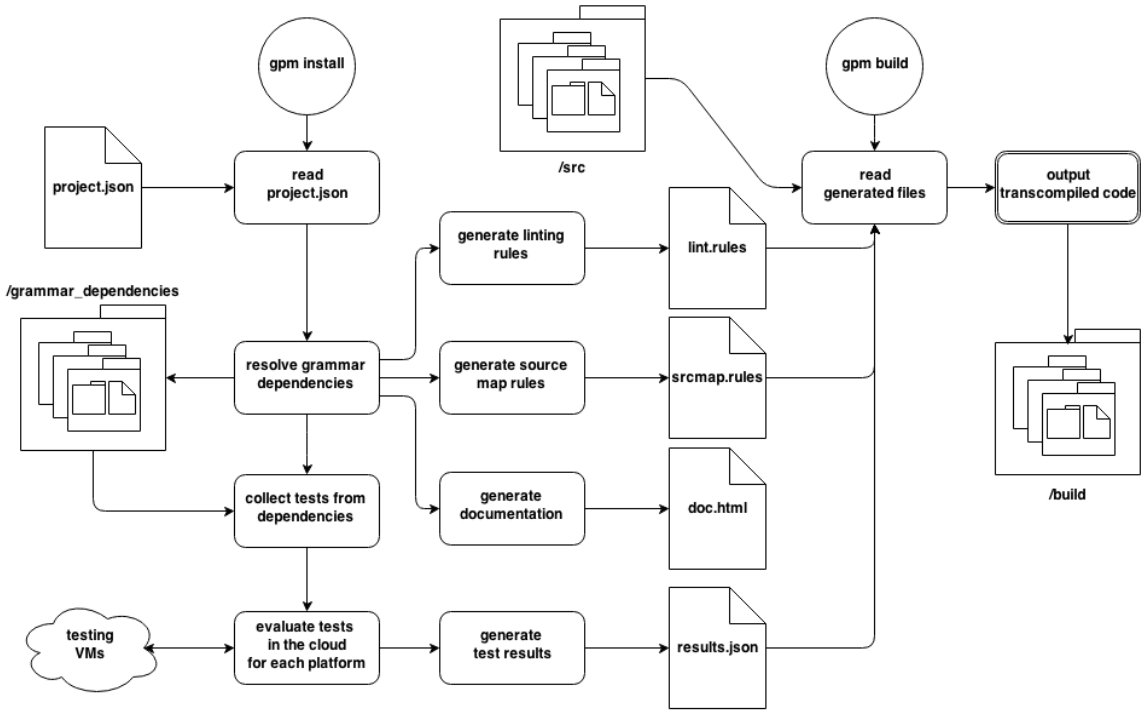

Fig. 2. GPM's high-level architecture.

Linting rules, although not mandatory, should be specified in order to flag suspicious language usage as well as to perform static analysis of the source code, avoiding known bad practices and catching bugs at build time. Source maps are a way to map a combined/minified file back to an unbuilt state, so that during runtime the line number of a fault is displayed correctly in relevance to the source file. In our case, sweet.js was able to generate source maps automatically by supplying it's transcompiler with the --sourcemap flag.

Support for various text editors and IDEs can be achieved by evaluating the contents of the generated linting rules. Projects hosted and maintained in Github and other open source repositories use a markdown format file to provide the documentation. The whole documentation produced by GPM is a concatenated version of each package's documentation.

In order to demonstrate the potential for syntactic expressiveness, we assembled three simple use cases (code available in [50]), paired with their expanded code. Fig.3.a illustrates a "fat arrow function", as specified in the ES6 Harmony work-in-progress, with an implicit return statement and `.this` bound to its original context. Fig.3.b demonstrates control flow of asynchronous code, using `await` as proposed for the upcoming ECMA7 specification [51]. The equivalent functionality can be produced using generators, with partial support and a considerable performance overhead when polyfilled. Finally, Fig.3.c illustrates the ability to implement a Domain-Specific Language (DSL) that affectively abstracts away language technicalities.

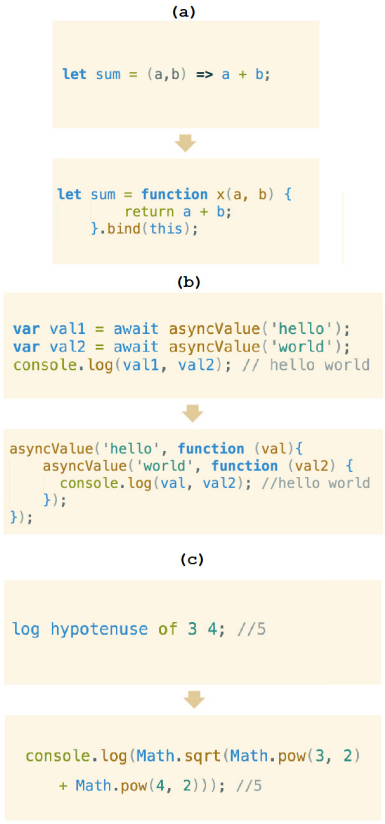

Fig. 3. Use cases a) fat arrow function (ES6) (b) await (ES7) (c) new DSL implementation.

V. Conclusions and Future Work

The introduction of GPM could solve the compatibility and complexity problems described in the paper. Languages that compile to JavaScript can be implemented in a modular way, and extended on demand, thus, allowing for the creation of new languages, both DSL and general purpose ones, that serve emerging trends and follow the evolution of the target language without suffering any of the drawbacks the target language may have. The packages that compose each language are published using an open source approach, they can be reused and optimized for performance as well as compatibility, while at the same time giving insight to committees specifying specs, about the current trends that should be taken into consideration to prioritize certain feature updates, ultimately tightening the feedback loop between the editors of web standards and web developers, aiding the beliefs of browser vendors and committee members presented in the Extensible Web Manifesto.

Furthermore, ES6 “Harmony” is greatly inspired by open source projects, so by creating an ecosystem supporting the committee working on the standards is expected to evolve both the language and the web at a rapid rate. Instead of building generic polyfills to address a number of cases and brace up for the frequent updates of JavaScript, compatibility can be achieved by bonding the compatibility mechanisms within each package. Hence developers of complex applications are empowered with expressiveness, to focus on the actual implementation part of their ideas and not suffer from dealing with quirks and inconsistencies from lack of normalization in the environments the code is to be run, eventually taking a quite a toll on their productivity.

The extension of GPM is in our future plans. As already mentioned, JavaScript implementations by browser vendors do not strictly follow the ECMA specifications and partially implement the current working draft in steps for developers to get acquainted with the changes and upcoming features. Thus, a program can be composed following different specifications in its parts. In order to further abstract the notion of valid specification, a cloud of virtual machines may be used to assist the transcompilation process. Optional unit tests can be collected from the dependency tree and then run on different platforms hosted in the cloud. The results can be used to optimize the transpiled code for each platform, for example if maps are supported the native implementation (i.e. Firefox 31.0) will be used, if an implementation is not available (Chrome 36.00) GPM would use a compatible transformation, if available. If the tests fail, the target platform will be marked as non compatible (i.e., Internet Explorer 11). The resulting code can be cached and served to same user-agents to maintain cross platform compatibility and automatically offer the performance and features of the latest JavaScript engines where applicable.

References

[1] https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[2] https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html

[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[4] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

[5] http://nodejs.org/

[6] https://www.npmjs.org/

[7] Azat Mardan, "Publishing Node.js Modules and Contributing to Open Source", Practical Node.js, Apress, pp 261-267, July 2014.

[8] http://www.modulecounts.com/

[9] I. K. Chaniotis, K.-I. D. Kyriakou and N. D. Tselikas, "Proximity: a real-time, location aware social web application built with Node.js and AngularJS", MobileWeb and Information Systems. Springer, BerlinHeidelberg, pp 292–295, 2013.

[10] I. K. Chaniotis, K.-I. D. Kyriakou and N. D. Tselikas, "Is Node.js a viable option for building modern web applications? A performance evaluation study", Computing, Springer, pp. 1-22, March 2014.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript", ECOOP 2010, Lecture Notes in Computer Science. Springer, 2010.

[12] http://kangax.github.io/compat-table/es5/

[13] Brendan Eich, "JavaScript: Taking both the High and the Low Roads", O'Reilly Fluent, The Web Platform, March 2014.

[14] https://github.com/tvcutsem/harmony-reflect

[15] https://github.com/jdarling/Object.observe

[16] https://github.com/google/traceur-compiler

[17] https://github.com/facebook/regenerator

[18] http://www.commonjs.org/specs/

[19] https://github.com/systemjs/systemjs

[20] https://github.com/substack/node-browserify

[21] https://github.com/amdjs/amdjs-api/blob/master/AMD.md

[22] https://www.npmjs.org/package/jquery

[23] http://bower.io/

[24] http://jamjs.org/

[25] http://jspm.io/

[26] http://duojs.org/

[27] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[28] http://facebook.github.io/react/index.html

[29] http://facebook.github.io/react/docs/jsx-in-depth.html

[30] http://gruntjs.com/

[31] http://gulpjs.com/

[32] https://github.com/broccolijs/broccoli

[33] http://coffeescript.org/documentation/docs/cake.html

[34] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

[35] http://www.typescriptlang.org/

[36] https://www.dartlang.org

[37] http://coffeescript.org/

[38] http://jsperf.com/fibgen/8

[39] http://code.google.com/p/traceur-compiler/

[40] https://www.npmjs.org/doc/cli/npm-shrinkwrap.html

[41] http://expressjs.com/

[42] https://github.com/facebook/regenerator

[43] Patrick Lee, "CoffeeScript in Action", Manning Publications, May 2014.

[44] http://koajs.com/

[45] https://github.com/visionmedia/co

[46] http://extensiblewebmanifesto.org/

[47] http://sweetjs.org/

[48] Guy L. Steele, "Common LISP: the language", Digital press, 1990.

[49] http://esprima.org/demo/parse.html

[50] https://github.com/J-Chaniotis/gpm-concept

[51] http://wiki.ecmascript.org/doku.php?id=strawman:deferred_functions#syntax