Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript

Diogo Costa Reis
ist187526
diogo.costa.reis@tecnico.ulisboa.pt

Instituto Superior Técnico
Av. Rovisco Pais, 1
1049-001 Lisboa
Tel: +351 218 417 000
mail@tecnico.ulisboa.pt

Abstract. Keywords: ECMAScript · Specification Language · Reference Interpreters · Test262

Table of Contents

1 Introduction

2 Goals

3 Background

This chapter provides an overview on the ECMAScript standard, the Test262 that are used to test the correct implementation of the ECMAScript standard, and finally an outline of the new metadata generated.

3.1 ECMAScript

JavaScript (JS) is a programming language mainly used in the development of client side web applications, also being one of the most popular programming languages. According to both GitHub and StakeOverflow statistics, JavaScript finished 2021 as second most active languages on GitHub[1] as well as on Stack-Overflow.[2]

ECMAScript standard[2] is the official document, written in the English language, in which the JavaScript language is defined. This document is in constant evolution, being updated by the ECMA Technical Committee 39 (TC39), which is responsible for maintaining the standard. The standard is currently in its twelfth version. The standard specifies the `JavaScript` language, to ensure its multiple compilers and interpreters implementations are coherent. Some of the `JavaScript` compilers are the Hop [1] and the JSC [5] compilers, the most popular interpreters are Node.js [**?**] and SpiderMonkey [3]. These are only four implementations among many others, which come along with the many use cases that `JavaScript` has. `JavaScript` is mostly used in the web context, both client-side within browsers and server-side, but also in embedded devices. Since `JavaScript` is used in so many scenarios and across so many different contexts, it is highly important that ECMAScript standard is defined in great detail to ensure consistency. Browsers, for example, need to run `JavaScript` implementations that coincide so that websites are correctly rendered and exhibit the same behavior. In order to achieve coherent implementations, the standard defines the types, values, objects, properties, syntax, and semantics of `JavaScript` that must be the same in every `JavaScript` compiler and interpreter, while allowing `JavaScript` implementations to define additional types, values, object, properties, and functions.

The `JavaScript` language can be divided into three major components, those being expressions and commands, built-in libraries, and finally internal functions.

– Expressions and commands describe the behavior of static constructions, detailing the semantics of the diverse expressions (e.g., assignment expressions,

[1] Second most utilized language based GitHub pull requests - https://madnight.github.io/githut/

[2] Tendencies based on the Tags used - https://insights.stackoverflow.com/trends

built-in operators, etc.), commands (e.g., loop commands, conditions command, etc.), and built-in types (Undefined, Null, Boolean, Number, String and Object).

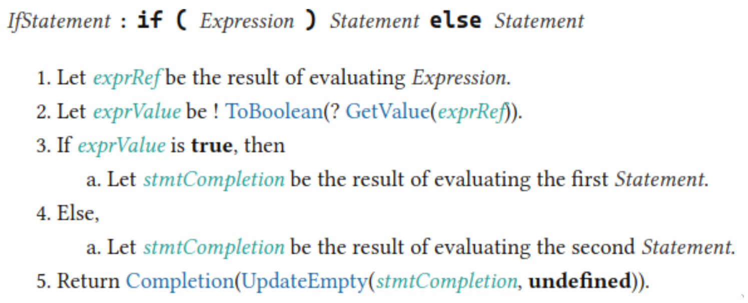
– The internal functions of the language are used to define the semantics for both expressions and commands, as well as the built-in libraries. Internal functions are not exposed beyond the internal context of the language. In other words, no JavaScript program uses internal functions directly.
– Finally, built-in libraries encompass all the internal objects available when a JavaScript program is executed. Internal objects expose many functions implemented by the language itself, including functions to manipulate numbers, text, arrays, objects, among other things.

The remaining subsection provides a description of the three types of artifact described in the standard.

Semantics of IF statement Figure 1 shows a snippet of the ECMAScript standard description of the `IF` command. In order to evaluate `IF` commands with the shape:

```
if (Expression) Statement1 else Statement2
```

the language begins by evaluating the `Expression` storing the result in the variable `exprRef` (step 1). The previous step will be used as Boolean, therefore, the result of the previous step will be converted to a Boolean using the internal functions `ToBoolean` and `GetValue`, and having the result stored in the variable `exprValue` (step 2).A different `Statement` will be followed depending on `exprValue`. If `exprValue` has the value `true` the variable `stmtCompletion` will have the evaluation of the first `Statement` (step 3). Otherwise, the variable `stmtCompletion` will store the result of evaluating the second `Statement` (step 4). Finally, a `Completion` will be returned, if the `stmtCompletion` has non empty value then it will be returned, however, when the value is empty it will be replaced with undefined (step 5).



IfStatement **:** **`if`** **`(`** *Expression* **`)`** *Statement* **`else`** *Statement*

```
1. Let exprRef be the result of evaluating Expression.
2. Let exprValue be ! ToBoolean(? GetValue(exprRef)).
3. If exprValue is true, then
      a. Let stmtCompletion be the result of evaluating the first Statement.
4. Else,
      a. Let stmtCompletion be the result of evaluating the second Statement.
5. Return Completion(UpdateEmpty(stmtCompletion, undefined)).
```

Fig. 1. ECMAScript definition of an if-else statement

Semantics of the Pop function The Array built-in is an object as any other in JavaScript. The main difference is in its properties. Array Objects have a property `length` that contains the size of the array, as well as a property for each element of the array (from zero to `length` minus 1).

Figure 2 shows a simplified version of an array performing the pop function, where `(a)` and `(b)` are the before and after respectively. Before preforming `pop` `(a)`, the array has three properties `length`, `0`, and `1`. Property `length` represents the size of the array that has value `2`, while the properties `0` and `1` store the first (`banana`) and second (`kiwi`) elements of the array respectively. After `pop` is preformed `(b)`, the last element is of the array is removed (highlighted in red at `(a)`) and the `length` property (highlighted in green) is decremented by one since the size of the array changes to one.

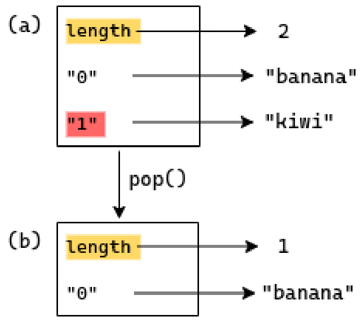
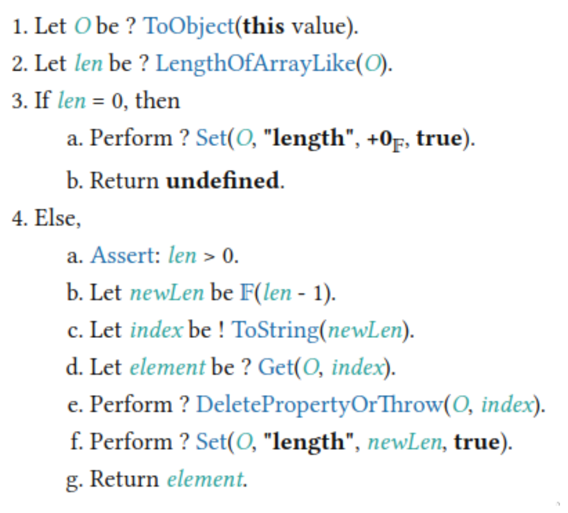


Fig. 2. Example Array.pop

Figure 3 shows a snippet of the ECMAScript standard description of the pop function in the Array Built-in. To begin with, the array will be converted to and Object using the `ToObject` function, and stored in the `O` variable (step 1). Afterwards, the array length of the previously calculated variable will be calculated with the `LengthOfArrayLike` internal function, and storing the result in the `len` variable (step 2). At this point there are to ways to proceed depending on the value of `len`. If the value is zero, the Array is empty, then the property `length` of `O` is set to zero and `undefined` is returned (step 3). Otherwise, when `len` is different from zero, meaning that the Array is not empty, the Array's last element will be removed (described in Figure 2) and returned (step 4). To begin with, the language will assert that `len` is positive (step 4.a). Afterwards, the `newLen` variable will store the value of `len` decremented by 1 (step 4.b). The variable `index` will store the variable calculated in the previous step represented as a String converted with the `toString` function (step 4.c). Then, stores the value of the `O` variable at the property corresponding to `index` in the `element` variable using the `Get` function (step 4.d). Subsequently, deletes the previously mentioned property of the `O` variable with the `DeletePropertyOrThrow` function (step 4.e). In addition, sets the `length` property of the `O` variable to the `newLen` using the `Set` function (step 4.f). Finally, returning the value of the variable `element` (step 4.g).



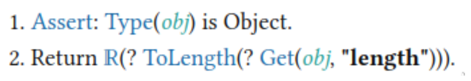
```
1. Let O be ? ToObject(this value).
2. Let len be ? LengthOfArrayLike(O).
3. If len = 0, then
    a. Perform ? Set(O, "length", $+0_{\mathbb{F}}$, true).
    b. Return undefined.
4. Else,
    a. Assert: len > 0.
    b. Let newLen be $\mathbb{F}$(len - 1).
    c. Let index be ! ToString(newLen).
    d. Let element be ? Get(O, index).
    e. Perform ? DeletePropertyOrThrow(O, index).
    f. Perform ? Set(O, "length", newLen, true).
    g. Return element.
```

Fig. 3. ECMAScript definition of Array.pop

LengthOfArrayLike internal function Figure 4 shows a snippet of the ECMAScript standard description of the `LengthOfArrayLike` internal function, that evaluates the function:

```
LengthOfArrayLike (obj)
```

The language starts by asserting that `obj` is an `Object` (step 1). Afterwards, gets the value of the property `length` from `obj` using the function `Get`. Then, converts the previously mentioned value to an Integer that represents the length with the `ToLength` function, and finally returns said Integer (step 2).



```
1. Assert: Type(obj) is Object.
2. Return $\mathbb{R}$(? ToLength(? Get(obj, "length"))).
```

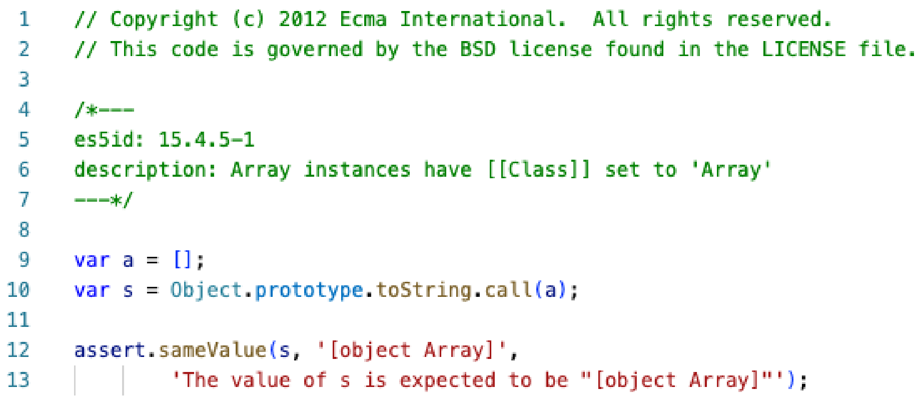
Fig. 4. ECMAScript definition of the LengthOfArrayLike

3.2 Test262

Implementing a `JavaScript` engine is particularly difficult since it involves dealing with the many corner cases that exist in the language. To test that corner cases are correctly dealt with there is `Test262`[4], the ECMAScript standard test battery. Although, `Test262` is vital to the `JavaScript` engines, it is very hard to maintain due it's complexity, the total number of tests is around 39837 divided into 87 subfolders, each correspond to roughly one section of the standard. `Test262` complexity grows with changes to the standard since in most cases backward compatible is maintained except for a few select cases.

Due to the ECMAScript standard being so extensive most implementations are only partial, especially implementations and analysis developed in academic contexts. In order to test partial implementations, one must be able to obtain the applicable set of tests from all the tests contained in Test262. Selecting the applicable tests is not a trivial matter because there are too many tests and too many features. The current methodology is that each development team manually selects the tests that are applicable to their corresponding implementation. This raises the problem that there is not standard and precise way of picking the all the right tests from the almost 40000 tests in `Test262`, making the possibility of human error when selecting the applicable tests likely.

Figure 5 shows a test from `Test262`. Every test of `Test262` has 3 parts: first is the copyright section represented with the comment `//` (lines 1 and 2), second is the `Frontmatter` section between `/*---` and `---*/` with some metadata about the test (lines 4 to 7), and finally is the `Body` section with the code of the test (lines 9 to 13). The copyright section has information about the owner and license of the test. The `Frontmatter` section has the test id (15.4.5-1) and a description of the test. Finally, the `Body`'s code tests the correct implementation of the standard.



```
 1   // Copyright (c) 2012 Ecma International.  All rights reserved.
 2   // This code is governed by the BSD license found in the LICENSE file.
 3
 4   /*---
 5   es5id: 15.4.5-1
 6   description: Array instances have [[Class]] set to 'Array'
 7   ---*/
 8
 9   var a = [];
10   var s = Object.prototype.toString.call(a);
11
12   assert.sameValue(s, '[object Array]',
13           'The value of s is expected to be "[object Array]"');
```

Fig. 5. Test262 es5id: 15.4.5-1

The `Frontmatter` has keywords to hold metadata of the test. These keywords are associated with specific elements of metadata concerning the test. Bellow is the list of possible keywords and their meaning:

- `description` - contains a short description about what will be tested;
- `esid` - contains the hash identifier of the ECMAScript portion associated with the feature that will be tested (the identifier references the most recent version of ECMAScript when the test is created);
- `info` - contains a deeper explanation of the test behavior, frequently includes a direct citation of the standard;
- `negative` - indicates that the test throws an error; associated to the keyword will be the type of error the test is supposed to be thrown (e.g. `TypeError`, `ReferenceError`) as well as the phase in which the error is expected to be thrown (e.g. `parse` vs `resolution` vs `runtime`);
- `includes` - contains the list of `harness` files that should be included in the execution of the tests (`Test262` makes use of a large number of auxiliary function defined in a dedicated library referred to as the `Test262 harness` described later in this section);
- `author` - contains the identification of the author of the test;
- `flags` - contains a list of booleans for each test property, the properties being: (1) `onlyStrict`, the test is only executed in strict mode; (2) noStrict, the test will only be executed in mode *sloppy*; (3) `module`, the test must be integrated as a `JavaScript` module; (4) `raw`, executes the test without any modification, which implies running as `noStrict`; (5) `async`, the test is contains asynchronous functions; (6) `generated`, the test generates the files specified by the property; (7) `CanBlockIsFalse` and (8) `CanBlockIsTrue`, the test will run if the property `CanBlock` of the `Agent Record` executing it is false and true respectively; (9) `non-deterministic`, indicates that the semantics used in the test are intentionally under-specified and therefore the test passing or failing should not be regarded as an indication of reliability or conformance;
- `features` - contains a list of features that are used in the test;
- `es5id` and `es6id` - indicates that the feature being tested belongs to ECMAScript 5 and 6 respectively and contains the hash identifier of the section of the standard it belongs to; these keywords have been deprecated and substituted by esid.

As Figure 5 illustrates, it is often the case that the metadata of a test is incomplete. Some tests also have the wrong metadata. As part of this thesis, we plan to process all the tests to check and correct their corresponding metadata as well as completing the metadata that is missing.

The example in Figure 5 has 2 keywords, description and the deprecated es5id. Besides the obvious upgrade from `es5id` to `esid` it would be useful to have `includes` with the `harness` files needed to execute the test. The `harness` information is very useful since it makes it easy to identify the part of the `harness` needed to run that test, opening the door for loading only part of the `harness` instead of the whole `harness` which is the current approach.

New Metadata Besides the metadata that Test262 tests currently include, it would be useful for them to have additional information regarding:

– `syntactic construct` - list of all syntactic constructions used in the test;
– `version` - the ECMAScript version of the standard in which the feature being tested was introduced;
– `built-ins` - list of all the built-ins used in the test.
– `harness-functions` - list of all the harness functions required to run the test.

This above metadata critical for filtering the tests when considering partial implementations of the language. For instance, if a JS engine only supports the 5th edition of the standard, one must be able to obtain all the corresponding tests for that `version`. Analogously, if a JS engine only implements certain `built-in` objects, one has to be able to filter out the tests that make use of the `built-in` objects that it does not implement. This would provide consistency and standardization to the selection of applicable tests to a partial implementation of the standard. As for the `harness-functions`, it provides important information about the functions of `harness` that are used in the test. That information is relevant because only a small part of the `harness` is need in each test even though the whole library is loaded and tested. The harness has a total of 7290 lines in 32 files and is tested by 96 tests. By only including the exact functions that a test requires, one can speedup the testing process significantly.

3.3 MetaData262 v0

This thesis continues the master thesis of Miguel Trigo[?], in which he developed a preliminary version of `MetaData262`. More specifically, he built a MongoDB database storing the metadata of all Test262 tests, representing the metadata of each test as a JSON object.

In `MetaData262 v0`, each test is associated with a JSON object storing the various metadata properties of the test and their corresponding values. Those metadata properties being the keywords of the `Frontmatter` mentioned in the previous section. Besides, the existing metadata properties, various new properties were added to `MetaData262 v0`. We can divide these new properties into 3 main groups: location, extra front matter, and statistics. Each of these groups will be discussed next.

Location group The location group contains information about the path to the test inside the `Test262`, having the property `path` associated to a string with the full path to the test.

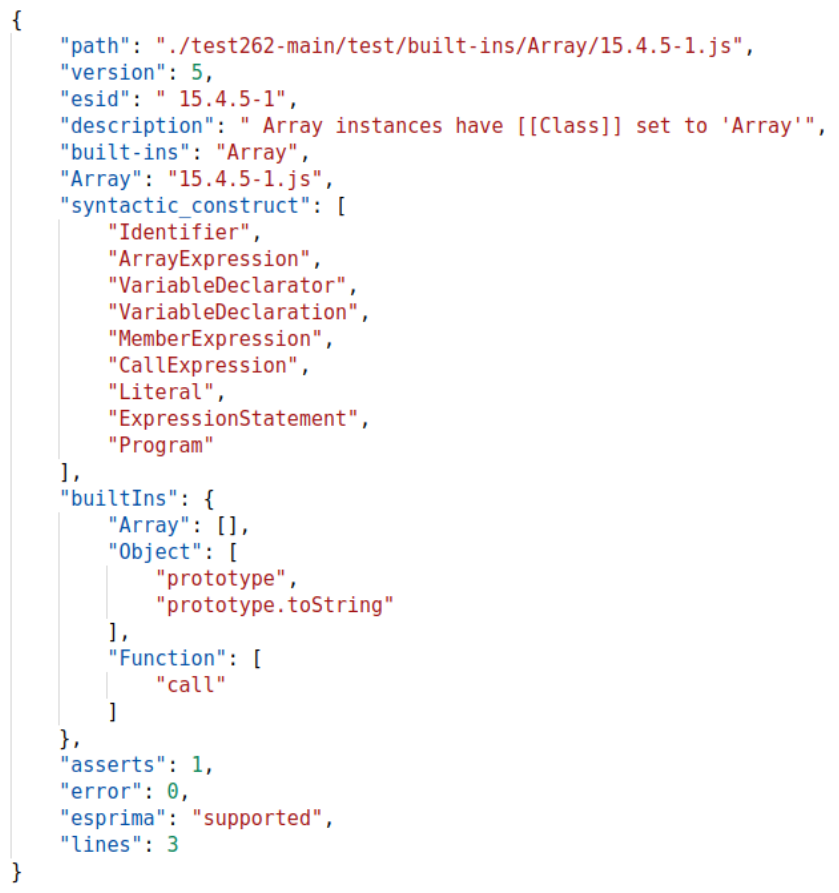
Extra front matter group The extra front matter group contains new metadata generated for `MetaData262 v0`. The new metadata generated is associated with the properties: `syntactic_construct`, `version`, and `builtIns` that were mentioned in the previous section as important metadata to add. `Version` stores the `ECMAScript` standard version the test belongs to. `Syntactic_construct` stores an array with all the syntactic constructions that the test contains. As for `BuiltIns`, it stores all the built-ins that test interacts with and associates with each built-in its fields and functions that are used.

Statistics group Finally, the statistics group contains some statistical data about the tests. The properties this group has associated are asserts, error, esprima, and lines. The asserts property holds the amount of assert statements in the test. The error property stores the amount of calls made to the Test262 error functions. The esprima property stores *supported* or *not supported* depending on whether Esprima[?] supports the test. Esprima is a standard-compliant ECMAScript parser that is also developed in ECMAScript, Esprima fully supports up to version 7 of the standard. The property lines holds the number of lines of code the test has, comments as well as empty lines are ignored.

Figure 6 shows the JSON object storing the metadata generated from the Test262 test given in Figure 5. The JSON object begins with the path property and the path to the test associated with it. After, the version property indicates that this test belongs to the fifth version of the standard, if the version is not able to be calculated the field will be omitted. Followed by properties in the test's Frontmatter (note that the deprecated es5id is replaced to the esid property). Next is the static_construct property that holds an array with the syntactic constructors of the test that were generated. The builtIns property holds a JSON object, in which, each property corresponds to a built-in and associated to it are its functions and fields that are interacted with in the test. Finally, the statistics group of properties. The asserts property that indicates that one assert statement is used in this test, the error holding zero since no error functions of the Test262 were used in this test, the esprima property that indicates this test is supported by Esprima, and the lines property indicates that this test has 3 lines of code (the example in Figure 5 splits the last line of the test in order to improve the readability of the figure hence only 3 lines are counted in the actual test).

As part of the setting up of MetaData262 v0, M. Trigo had to compute the missing Frontmatter properties as well as the values of the new properties that he introduced. In most cases, the computation was straightforward, only involving a simple syntactic analysis of each test. However, the computation of the version and the built-ins used in each test was more involved and we will go through it in more detail below.

Calculation of Version metadata The version of the standard a test belongs to is calculated using 3 different approaches: dynamic, static, and hybrid. The dynamic approach is based on the waterfall model, running the tests in various JS engines, each corresponding to a specific version of the standard. As Figure 7 illustrates, each test starts by running in the JS engine corresponding to version 5 of the ECMAScript standard (represented as ES). If the test output is correct then the test belongs to version 5 of the standard, otherwise the test will be run in the engine implementing the next version of the standard. This process repeats until the last engine, associated with version 11 of the standard, if the test output is not the expected again then the test is not supported. The static approach evaluates the syntactic tree of the test generated by Esprima, a widely used JS parser, as a JSON object. Starts by finding all the JavaScript keywords in the AST. Then associates each JS keyword with the version of the standard it



```
{
    "path": "./test262-main/test/built-ins/Array/15.4.5-1.js",
    "version": 5,
    "esid": " 15.4.5-1",
    "description": " Array instances have [[Class]] set to 'Array'",
    "built-ins": "Array",
    "Array": "15.4.5-1.js",
    "syntactic_construct": [
        "Identifier",
        "ArrayExpression",
        "VariableDeclarator",
        "VariableDeclaration",
        "MemberExpression",
        "CallExpression",
        "Literal",
        "ExpressionStatement",
        "Program"
    ],
    "builtIns": {
        "Array": [],
        "Object": [
            "prototype",
            "prototype.toString"
        ],
        "Function": [
            "call"
        ]
    },
    "asserts": 1,
    "error": 0,
    "esprima": "supported",
    "lines": 3
}
```

Fig. 6. Metadata generated for test esid: 15.4.5-1.js

was introduced in. Finally, the keyword with the most recent version associated will dictate the test version. For example, if the test has only two keywords associated with version 6 and 8, then the test will belong to the version 8 of the standard. The hybrid approach is calculated using the results of the dynamic and static approaches. The hybrid approach merges the results by maintaining the higher version detected between each approach for each test. For instance, if the dynamic approach's result for a test is version 6 and the static's result for the same test is version 8, then, the test with belong to the version 8 of the standard according to the hybrid approach.

Calculation of built-ins metadata The `built-ins` used by each test are calculated using two separate approaches: dynamic and static. As in the calculation if the `ECMAScript` version both approaches are combined to improve the results. The static approach makes use of syntactic tree generated by `Esprima` as

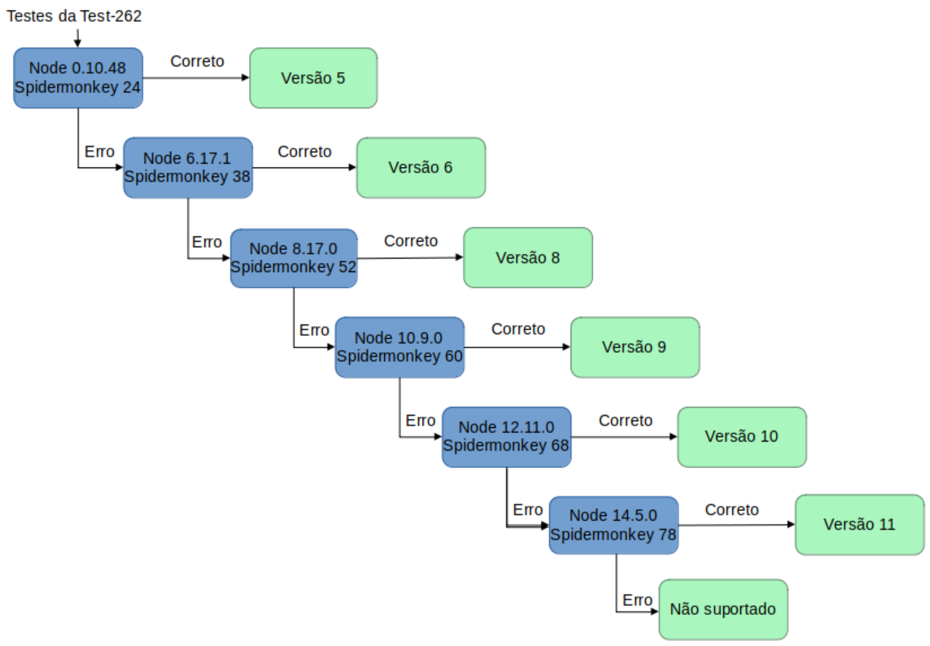

Testes da Test-262
Node 0.10.48 Spidermonkey 24
Correto
Versão 5
Erro
Node 6.17.1 Spidermonkey 38
Correto
Versão 6
Erro
Node 8.17.0 Spidermonkey 52
Correto
Versão 8
Erro
Node 10.9.0 Spidermonkey 60
Correto
Versão 9
Erro
Node 12.11.0 Spidermonkey 68
Correto
Versão 10
Erro
Node 14.5.0 Spidermonkey 78
Correto
Versão 11
Erro
Não suportado

Fig. 7. Waterfall model of the dynamic approach to calculate the ECMAScript version

a JSON object from the test code. Next, traverses the AST to find expressions that interact with built-in objects. In addiction to the built-in, the fields and functions will also be outputted. As for the dynamic approach, it makes use of the built-ins functions in order to log whenever the function is called. In essence, a wrapper was created for each function that replaces it, the wrapper first logs that it was called, and then preforms the original method with the arguments that were supplied.

4 Related Work

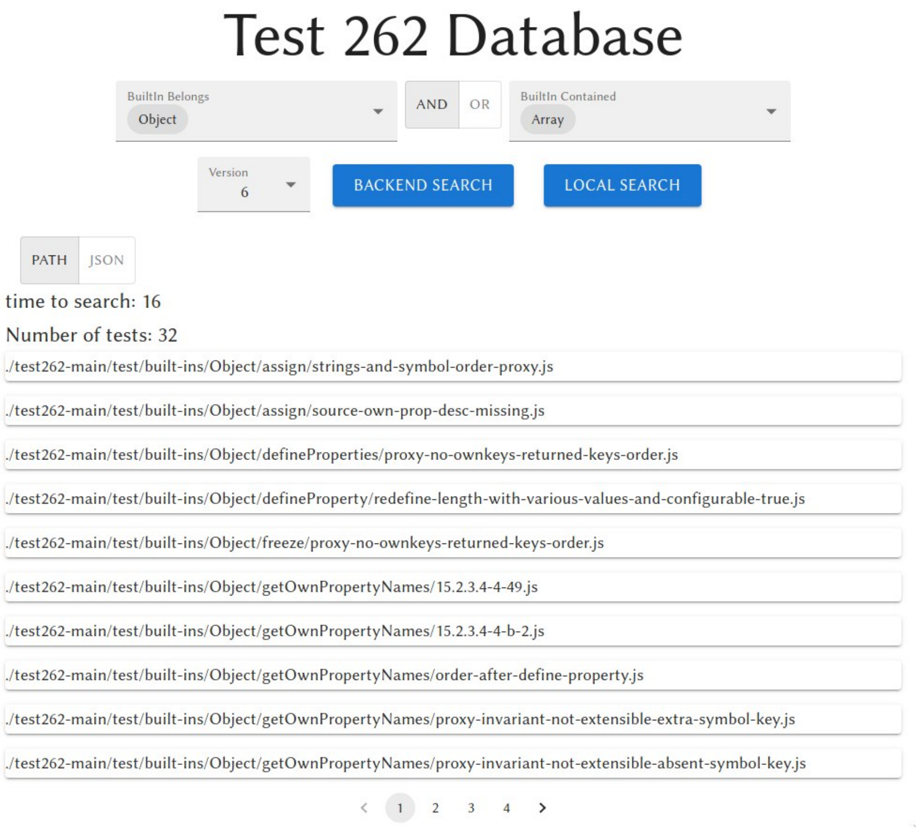
5 Design and Methodology

6 Evaluation and Planning

7 Conclusion

References

1. Ambiente de programação multi-tier para o javascript, https://github.com/manuel-serrano/hop, acedido a 2020-12-21



time to search: 16

Number of tests: 32

```
./test262-main/test/built-ins/Object/assign/strings-and-symbol-order-proxy.js
./test262-main/test/built-ins/Object/assign/source-own-prop-desc-missing.js
./test262-main/test/built-ins/Object/defineProperties/proxy-no-ownkeys-returned-keys-order.js
./test262-main/test/built-ins/Object/defineProperty/redefine-length-with-various-values-and-configurable-true.js
./test262-main/test/built-ins/Object/freeze/proxy-no-ownkeys-returned-keys-order.js
./test262-main/test/built-ins/Object/getOwnPropertyNames/15.2.3.4-4-49.js
./test262-main/test/built-ins/Object/getOwnPropertyNames/15.2.3.4-4-b-2.js
./test262-main/test/built-ins/Object/getOwnPropertyNames/order-after-define-property.js
./test262-main/test/built-ins/Object/getOwnPropertyNames/proxy-invariant-not-extensible-extra-symbol-key.js
./test262-main/test/built-ins/Object/getOwnPropertyNames/proxy-invariant-not-extensible-absent-symbol-key.js
```

Fig. 8. Website for searching the metadata in construction

2. especificação da linguagem ecmascript® , 6.0 edition / june 2015, http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf, acedido a 2020-12-21
3. Spidermonkey is mozilla's javascript engine written in c and c++, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, acedido a 2020-12-21
4. Test262 - official ecmascript conformance test suite, https://github.com/tc39/test262/, acedido a 2020-06-07
5. Um motor javascript que corre em cima do motor javascript do browser, https://github.com/mbbill/JSC.js, acedido a 2020-12-21