A Reference Implementation of ES6 Built-in Libraries

Jorge Brown

***Abstract*—JavaScript is the *de facto* language for implementing client side Web applications. It is specified in the ECMAScript standard, a long and complex document written in English that is updated with every new iteration of the language. Despite its popularity, JavaScript is not always an easy language to understand and its dynamic paradigm makes it hard to statically analyse. ECMAScript reference interpreters are artifacts produced to reason about the language in a controlled environment. To this end, we will leverage the ECMA-SL project, a research effort at IST whose goal is to build an executable version of the standard. Currently, the ECMA-SL project comes with an interpreter, ECMARef5, for the 5th version of the standard, which is now at its 12th version. We plan to assist with the transition of ECMARef5 to the 6th version of the standard, by aiding in the implementation effort of the built-in libraries of the ECMAScript 6 Standard. Alongside other strategies employed in the ECMA-SL project, to guarantee the quality of our implementation we test it against Test262, the official ECMAScript conformance test suite.**

***Index Terms*—JavaScript, regular expressions, continuation-passing style, debuggers**

I. Introduction

JavaScript is one of the most used programming languages in the world and the most commonly used to develop client-side Web applications, such as e-mail clients or online banking platforms, and so the Web has a big dependence on it. Recently it also has been gaining traction as a language for server-side scripting, specially with the Node.js [18] runtime, and even as a language for the development of various desktop applications, mainly using the Electron framework [11], e.g. Discord [10] and Visual Studio Code [24]. However, these applications do not all use the same JavaScript engine. Although a lot of them do use Google's V8 [7] JavaScript engine, there are other alternatives and it is critical that they all behave in the same manner. Should this not be the case, not only is it possible that some JavaScript applications do not execute properly in certain runtime environments, but it could also lead to critical security flaws. In order to mitigate this possibilities, it was deemed that a standardization of the JavaScript language was necessary and so the ECMAScript standard was created.

The ECMAScript standard (ES) [2] consists of the specification of the ECMAScript language's syntax and semantics. The following of the standard's specification by every JavaScript engine, should mean that they all behave in the same manner regardless of how they are implemented. This means that developers that build applications using the ECMAScript language can be confident that their application will behave as expected, independently of the engine it is run on. The ECMAScript standard is a long document written in English that describes the behavior of the language as if it was the pseudo-code of an interpreter, giving detailed steps on how to interpret each instruction. Over the years it has evolved substantially, regularly increasing in its size and detail. Figure 1 illustrates this evolution and shows us that sometimes the standard can double in size in a single iteration. Despite the large amount of iterations, the standard is not easy to maintain or alter, with the process of adding features being extremely complex, requiring new features to uphold the invariants of the language's semantics and maintain backwards compatibility, guaranteeing that the behaviour of older features remains unchanged. The document is managed and maintained by the TC39 committee [22] which is composed of JavaScript developers, browser representatives, academics, etc. As of now, the committee has a well established methodology used to extend and update the standard's specification called "The TC39 Process" [23]. This process is divided into 5 stages going from coming up with ideas of new features for the ECMAScript language, to selecting the best ideas and what challenges they may entail, to describing their syntax and semantics, refining them and eventually adding them to the standard in a polished state.

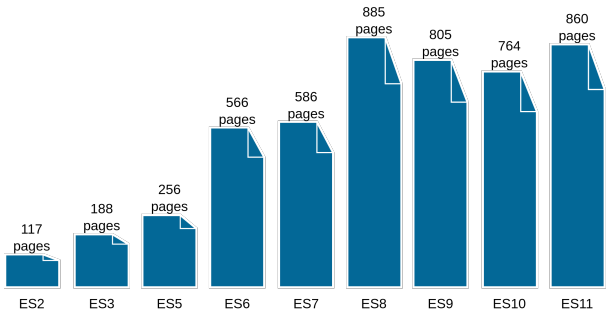

Fig. 1. Evolution of the number of pages describing the ECMAScript standard over time.

Naturally, not all implementations of the ECMAScript language follow the standard's described behavior exactly. They can be built using different programming languages and it may be convenient to represent some components of the standard in a different, although equivalent, manner. Most of the discrepancies are due to performance reasons. In particular, the industrial JavaScript implementations use JIT (just-in-time) compilation to guarantee the performance of their applications. If the JavaScript implementations do not follow the standard exactly, then how can it be determined that they are ECMAScript compliant? Currently, this is done through

exhaustive testing. Alongside the ECMAScript standard, the TC39 committee also maintains an official test suite called Test262 [6] whose purpose is to measure the conformity of a JavaScript interpreter to the official ECMAScript specification. However, testing is an incomplete method for determining this, as there have already been multiple bugs found in JavaScript engines that were not discovered by the test suite [8].

An alternative methodology is to maintain a reference implementation that follows the standard line-by-line and use this implementation has an oracle to test the conformity of other implementations of the language. This could be done by comparing the behavior of those implementations with the behavior exhibited by the oracle on concrete programs. In this sense, multiple academic projects have cropped up with the intent of producing a reference interpreter for JavaScript [8], [9], [12], [13], [15], [19], [20]. However, most of these reference implementations do not support the standard' built-in libraries, with those that partially do it only doing it at a very small scale. In fact, we can fairly say that most ECMAScript reference interpreters ignore the language's built-ins. However, they are an essential part of the language and their testing alongside the core of the language is critical to fully understand and reason about JavaScript implementations.

With the goal of filling the void left by the absence of a complete reference interpreter the *ECMA-SL project* [14], [16], [21] was created. The main goal of the project is to maintain a complete (with built-in support) executable specification of the standard written in an intermediate language specifically designed to do so, the ECMA-SL language, which stands for ECMAScript-Specification Language. The semantics and algorithms described in the ECMAScript standard are written as if they were the pseudo-code of a reference interpreter (executable specification). The ECMA-SL language mimics this pseudo-code, making the specification of a reference interpreter, almost a work of copying the semantics already described in the English standard. From the specification of the standard in ECMA-SL, it is possible to create numerous other artifacts, namely a natural language version of the standard structured as an HTML document, similar to the original. Currently, the ECMA-SL project includes a full specification of ES5 [3] and a partial implementation of ES6 [2]. Also included in the ECMA-SL project, are multiple tools in development that use the interpreter for many other purposes as Figure 2 shows. Some of the most relevant of them are HTML2ECMA-SL [14] and ECMA-SL2English [16], whose purpose is, respectively, to convert the standard's HTML document written in English to ECMA-SL code and to convert the executable specification of the standard back to its original form in natural language.

The goal of this paper is to build upon what already exists in the ECMA-SL project. We will extend the ECMARef6 reference interpreter, the executable specification of ES6, increasing its coverage of the standard. This will be achieved through the implementation of `Symbol` built-in library of ES6, which introduces an entirely new type of property key, and its dependencies using a line-by-line strategy to guarantee

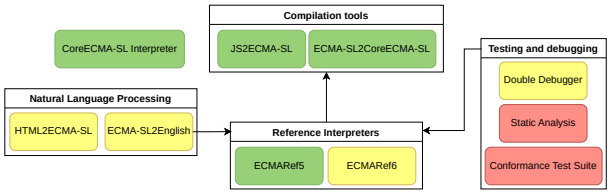

Fig. 2. Currently available (green), in development (yellow) and future (red) tools of the ECMA-SL project. Arrows signify dependency.

that our implementation is correct.

From a technical point of view, the implementation of this library posed a few challenges. In order to successfully implement the `Symbol` library, it was necessary to change the internal representation of ECMAScript objects in ECMA-SL as the previous representation was not fit to handle `Symbol` property keys.

In order to demonstrate that we attained our goals, we tested our reference implementation (ECMARef6) against the official test suite, placing special emphasis on the tests that target the 6th version of the standard and the built-in libraries. Overall, we obtained a 93.97% pass rate over all the built-in libraries and 95.44% over the ones we have emphasized. Although these results are far from 100%, the large majority of the failing tests, fail due to incomplete features in the core of the language and not in the built-in implementation.

This paper is structured in the following way: in Section II we present and discuss other projects related to the work done in the context of this thesis, including other reference implementations; in Section III we will present our analysis of the standard's specification as well as our implementation in ECMARef6 of the `Symbol` built-in library; in Section IV we present our evaluation methodology in more detail as well as our results; finally in Section V we go over some conclusions on the work performed as well as possible future endeavours.

II. Related Work

There have been numerous works on the development of reference interpreters for JavaScript. We will first look at the common trends amongst them, their coverage of the standard and their evaluation results. Then we discuss these works individually, looking at their most relevant design decisions and how they compare to the ECMA-SL project.

The need for a viable operation model of JavaScript was identified in 2008 with the work of Maffeis et al. [17]. Ever since, multiple other formal models of the JavaScript language have been written in various diverse languages such as Coq [1], K [4] and OCaml [5]. As these models appeared and took ideas from their predecessors, some concepts prevailed:

1) The formal model should be executable;
2) The formal model should pass the tests of the Test262 test suite;
3) A line-by-line strategy should be followed in order for it to be as identical as possible to the specification.

Consider Table I that compares the various existing formal models relative to the concepts mentioned above. Here we can

see that more recent models tend to follow the standard line-by-line, which is a good indicator that this methodology is effective. More recent models also tend to design their own DSLs (Domain-Specific Languages), like ECMA-SL, meant for implementing reference interpreters of the ECMAScript standard.

Although the wide adoption of these concepts results in more robust reference interpreters, there are still no models that offer significant support for the ECMAScript built-in libraries. However, their implementation is critical, for example, to reason about or test the implementation of the built-in libraries of JavaScript engines. Overall, since most ECMAScript programs use at least some of the libraries, not implementing them greatly reduces their utility. Table II compares the various models relative to the built-in libraries they implement, showing us that they are mostly ignored.

III. Symbol ECMAScript Specification and ECMA-SL Implementation

In this section, we present the ECMAScript specification and our ECMA-SL implementation of the `Symbol` built-in library. The purpose of the `Symbol` library is to provide another type of property key besides `string` values.

A. Examples

To demonstrate the use of `Symbol` values consider Listing 1. In this code-snippet, we see the use of the `Symbol` constructor to create two separate `Symbol` values. Although the arguments used in their creation were identical, they are still distinguishable. In line 4 we create an ECMAScript object using the object literal expression and then use our `Symbol` values `sym1` and `sym2` to add some properties to the newly created object. We associate the `string` `"xpto"` with the property key `sym1`. On the left-hand side of the assignment expression, we use bracket notation, as that is the only way to use `Symbol` values. The final instruction of our code-snippet does another assignment, this time using the other `Symbol` value. Since both the `Symbol` values were created with the same arguments, one may think that this last instruction essentially overwrites what was done in the previous one, by replacing `"xpto"` with `"abc"`. However, since `Symbol` values are always unique from each other, the last instruction will instead create a new property associated with the key `sym2`. In the end, we end up with an object we two properties: one associated with the `sym1` key; and another associated with the `sym2` key.

Listing 1. Example of the creation and use of `Symbol` values as property keys.

```
1 var sym1 = Symbol("example");
2 var sym2 = Symbol("example");
3
4 var obj = {};
5
6 obj[sym1] = "xpto";
7 obj[sym2] = "abc";
```

[1]Some reference interpreters focus only on the tests their implementation covers and therefore have a much higher passing percentage.

B. ECMAScript Specification

The entry point of the `Symbol` library in the ECMAScript specification is the `Symbol` constructor. This constructor creates `Symbol` values, which are primitives, unlike most constructors that create objects. In this subsection we first explain the difference between `Symbol` values and objects and how each is created and used. Afterwards, we present the `Symbol` constructor and its named properties, which are the main contribution of the `Symbol` library to the ECMAScript language. Finally, we dive into some of the methods available to `Symbol` objects through their prototype object.

a) Symbol Values: A `Symbol` value is a "primitive value that represents a unique, non-String Object property key". They are immutable and, even though they are not objects, have an internal property called `[[Description]]` that can be either `undefined` or a `string` value. In order to visualize their representation consider Figure 3. We start by creating two `Symbol` values using the same `Symbol("example")` expression. Note that the `new` keyword is not used when creating `Symbol` values and using it will throw a `TypeError` exception. Looking at the diagram, they look indistinguishable, except for the fact that they are two separate identical values. However, the `sym1 !== sym2` comparison in line 3 evaluates to `true`. This is because the way the specification defines the comparison of `Symbol` values is exactly by asserting if they are the same value, if the `sym1` and `sym2` variables point to the same memory addresses.

```
1 var sym1 = Symbol("example");
2 var sym2 = Symbol("example");
3 sym1 !== sym2; // true
```

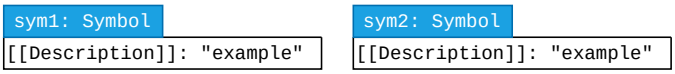

Fig. 3. Two `Symbol` primitive values.

b) Symbol Objects: As we have seen, `Symbol` values are primitive values and not the typical objects used in the rest of the built-in libraries. They have no `[[Prototype]]` internal property or named properties, so what is supposed to happen when the expression `Symbol("xpto").toString()` is evaluated? Using dot notation to access a property creates what is called a `Property Reference`. These have two components: (1) the expression before the dot, called the *base*; and (2) the expression after the dot, called the *reference name*. When the `base` of a `property reference` is not an object, it is converted to one by calling the `ToObject` internal operation of the ECMAScript standard. The `ToObject` internal operation is the only way to create `Symbol` objects. To compare `Symbol` values and objects, consider the code-snippet and diagram in Figure 4. In the first line of code, we create a `Symbol` value, using the constructor without the `new` keyword, with `"example"` as the value of `[[Description]]`. In the second line, we create the `Symbol` object by executing a type conversion. While using the `Object` constructor with the `new` keyword creates an

Reference interpreter	ES version	Exe.	# of tests passed	# of tests	Pass Rate	Implementation Language	L-B-L Strategy
S5	5	✓	8157	11275	72.35%	S5 Core Language	×
JSExplain	5	✓	>5000	11275	44.35%	Subset of OCaml	✓
KJS	5	✓	2782	11275	24.67%	K	×
JSRef	5	✓	3749	11275	33.25%	Coq	✓
JS-2-JSIL	5	✓	8797	11275	78.02%	JSIL	✓
ECMARef5	5	✓	9556	11275	84.75%	ECMA-SL	✓
ECMARef6	6	✓	18087	21662	83.50%	ECMA-SL	✓
JISET	10	✓	18064	29878	60.46%	IR_{ES}	✓

TABLE I
REFERENCE INTERPRETERS AND THEIR ADHESION THE VARIOUS STRATEGIES. (L-B-L SIGNIFIES LINE-BY-LINE)

Reference Intepreter	Object	Function	Boolean	Symbol	Error	Number	Math	Date	String	RegExp	Array	JSON
KJS	✓	✓	✓	×	✓	✳	×	×	✳	×	✳	×
JSRef	✓	✓	✳	×	×	✳	×	×	✳	×	×	×
JS-2-JSIL	✓	✓	✳	×	✳	✳	×	×	✳	×	×	×
ECMARef5	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
ECMARef6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

TABLE II
BUILT-IN SUPPORT BY THE OTHER REFERENCE INTERPRETERS. (✳ SIGNIFIES PARTIAL IMPLEMENTATION)

object, calling it as a function makes it so the argument is converted to an object by using the `ToObject` internal operation. In the diagram, we can see that the conversion to an object, creates a wrapper that stores the `Symbol` value in its `[[SymbolData]]` internal property. This wrapper is an ordinary object that also has a `[[Prototype]]` property that allows the `Symbol("xpto").toString()` expression to execute successfully.

```
1 var symValue = Symbol("example");
2 var symObject = Object(sym);
```

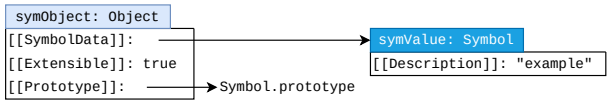

Fig. 4. Representation of a `Symbol` object (left) and `Symbol` value (right).

c) Symbol Constructor: The `Symbol` constructor, like all other constructors, has two important components: (1) the structure of the object that represents the constructor function with all its internal and named properties; and (2) the pseudo-code description of its behaviour. Consider the ES6 excerpt in Figure 5 that contains the descriptions of the `Symbol` constructor function. As we had noted, using the `new` keyword will make it so the value of `NewTarget` is not `undefined`, causing a `TypeError` exception to be thrown. Then the `description` argument is converted to a `string` value using the `ToString` internal operation. Finally, the `string` is used to create the `Symbol` value that is returned.

19.4.1.1 Symbol ([description])

When **Symbol** is called with optional argument *description*, the following steps are taken:

1. If NewTarget is not **undefined**, throw a **TypeError** exception.
2. If *description* is **undefined**, let *descString* be **undefined**.
3. Else, let *descString* be ToString(*description*).
4. ReturnIfAbrupt(*descString*).
5. Return a new unique Symbol value whose [[Description]] value is *descString*.

Fig. 5. ES6 description of the `Symbol` constructor function.

Although the `Symbol` constructor allows the creating of new `Symbol` values, most ECMAScript programs do not need to create new ones and just use ones that are made available through the named properties of the constructor. Consider the object diagram in Figure 6 where the constructor is displayed alongside some of its named properties, namely `iterator` and `toPrimitive`. These properties' descriptors are immutable, as all their properties that are not `[[Value]]` are set to `false`. Their `[[Value]]` property points to `Symbol` values that are created before any ECMAScript code is executed. There are 11 of these `Symbol` values made available through the named properties of the constructor `Symbol`. They do not allow any behavior that was not possible before, but it makes it unequivocal the type of property being accessed. For example, the `TypedArray` prototype object makes an iterator method available through the `TypedArray.prototype[Symbol.iterator]` property instead of the `TypedArray.prototype.iterator` property. But if it did use the `iterator string` value as the property key, it would still retain its functionality, changing

only the way it is accessed.

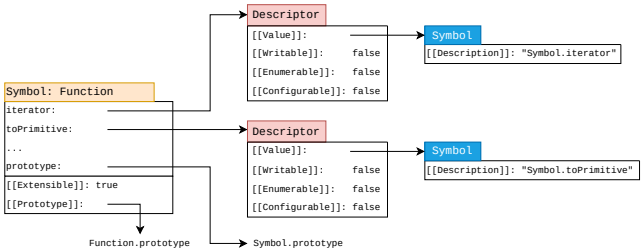

Fig. 6. Representation of the `Symbol` constructor object.

d) Symbol.prototype Object: Since `Symbol` values have no prototype and `Symbol` objects are only relevant in niche cases, the `Symbol.prototype` object does not have many methods. It has the three following methods:

1) `toString` - Returns the `string` value resultant from the concatentation of `"Symbol ("`, the symbol's `[[Description]]` and `")"`.
2) `valueOf` - If the `this` value is a `Symbol` object it returns `this.[[SymbolData]]`. If the `this` value is a `Symbol` value, it simply returns that value.
3) `Symbol.toPrimitive` - This method does the exact same as the `valueOf` method.

C. ECMA-SL Implementation

We are now in a position to introduce and explain our implementation of the `Symbol` built-in library in ECMARef6. First, we present the internal representation of `Symbol` values and objects. To conclude, we explain the changes that needed to be made to the structure of ECMA-SL objects to support the use of `Symbol` values as property keys.

a) Symbol Values: In ECMARef6, `Symbol` values are represented using ECMA-SL objects. Contrary to the ECMA-SL objects used to represent ECMAScript objects, `Symbol` values in ECMA-SL do not have a `JSProperties` property. Recall that a characteristic of `Symbol` values was that they were unique and compared through their values and not the value of their `[[Description]]`. In ECMA-SL, two `Symbol` values can be compared correctly using the `=` operator. However, it was still necessary for us to add the `ID` property to distinguish between `Symbol` values. The `ID` property holds an integer value that is different for every `Symbol` value. Why this was necessary, will be addressed later in this subsection. As an illustration of the structure just described, consider the object diagram in Figure 7. Here, the `Symbol` values created can be distinguished by their `ID` property, which has different values.

b) Symbol objects: In the case of `Symbol` objects, they now have their own `JSProperties` property, although it is not populated. The object structure of `Symbol` objects can be seen in Figure 8, where we can see that the `SymbolData` property points to an ECMA-SL object that represents a `Symbol` value and its `Prototype` property points to the `Symbol.prototype` object.

```
1 var symValue1 = Symbol("example");
2 var symValue2 = Symbol("example");
3 symValue1 === symValue2; //false
```

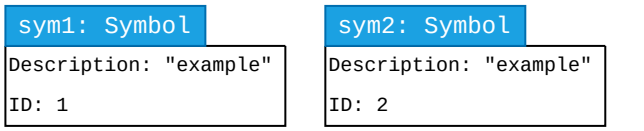

Fig. 7. Symbol values are unique and distinguished by their `ID` property.

```
1 var symValue = Symbol("example");
2 var symObj = Object(symValue);
```

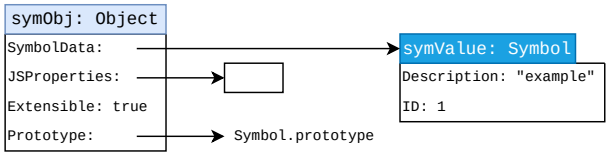

Fig. 8. Internal representation a `Symbol` object and value.

c) Symbol Values As Property Keys: Of particular importance is the actual use of symbols. Symbols are meant to be used as property keys alongside strings. Since the `Symbol` built-in library was introduced in ES6, the previous version of ECMARef, ECMARef5, had no support for `Symbol` values. Therefore, it was necessary to adapt the reference interpreter to accommodate this new property-key type.

ECMA-SL objects are collections of string-value pairs, meaning that it was not possible to just use the `Symbol` values as keys for our internal objects. Our first alternative was to just use the `string` value in the `Description` property of `Symbol` values. An example of this configuration can be seen in Figure 9 where an Object `a` and a `Symbol` value `sysValue` with `Description` `"example"` are created. In line 4, we assign the value `"b"` to object `a`, using the `string` value `"example"` as property key. The effect of this line can be seen in the `JSProperties` object, where there is a `property descriptor` with value `"b"` mapped to the property `example`. In line 5, we do a similar process but use `symValue` as the property key and set the value to `"c"`. We can see that another `property descriptor` with `Value` `"c"` is associated with the property `Symbol(example)`.

```
1 var a = {};
2 var symValue = Symbol("example");
3
4 a.example = "b";
5 a[symValue] = "c";
```

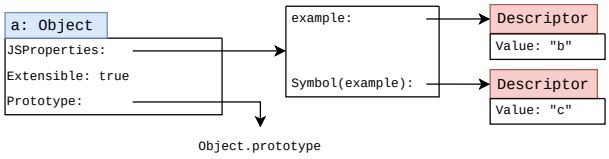

Fig. 9. Object property assignment using `string` and `Symbol` values.

However, this represents two big problems:

1) how do we distinguish between `Symbol` value keys with the same `Description` value;
2) how can we distinguish between the string `"Symbol(example)"` and the a `Symbol` value with `Description` `"example"` when they are both used as property keys.

The example in Figure 10 shows this exact scenario where the assignment done in line 6, is overridden by the ones in line 7 and 8, which is not the intended behavior. The intended behavior is that the assignments in lines 6, 7 and 8 all create their own `property descriptors`.

```
1 var a = {};
2 var symValue1 = Symbol("example");
3 var symValue2 = Symbol("example");
4
5 a.example = "b";
6 a[symValue1] = "c";
7 a[symValue2] = "d";
8 a["Symbol(example)"] = "e";
```

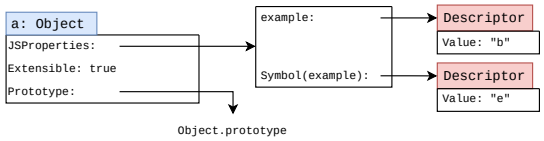

Fig. 10. Internal property storage design that does not allow for proper `Symbol` property keys integration.

In order to solve these issues, a new property called `JSPropertiesSymbols` was added to our ECMA-SL representation of ECMAScript objects. This property is meant to hold the named properties of the ECMAScript object that use `Symbol` value keys, while `JSProperties` now holds only the named properties that use `string` value keys. This solves the issue of using `string` values that match the `Description` of `Symbol` values. However, at this stage, this solution still suffers from collisions of `Symbol` values with identical `Description` values. To fix this issue, instead of the properties of the `JSPropertiesSymbols` object being the `Description` of the symbols, they are now the `ID`. The `ID` property of `Symbol` values is converted to a `string` and used as a key (recall that ECMA-SL objects are `string`-value pairs) to allow the distinction between all symbols. This is illustrated in Figure 11, where the `a` object now has a `JSPropertiesSymbols` property that points to an object that maps the IDs 1 and 2 of the symbols to the appropriate `property descriptors`. Although the current solution is enough to handle any reading or writing of properties using`Symbol` values in objects, an additional property `[[SymbolKeys]]` had to be added. This property allows the mapping of `ID` values to the correspondent `Symbol` values, enabling the retrieval of both the `Symbol` and `string` property keys of an object when methods like `Object.keys()` are called.

```
1 var a = {};
2 var symValue1 = Symbol("example");
3 var symValue2 = Symbol("example");
4
5 a.example = "b";
6 a[symValue1] = "c";
7 a[symValue2] = "d";
8 a["Symbol(example)"] = "e";
```

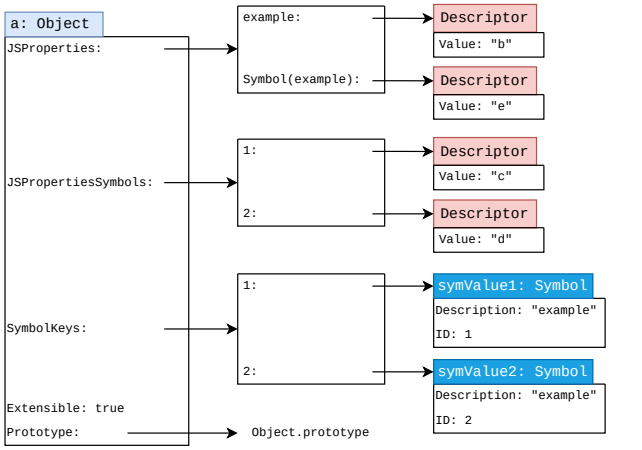

Fig. 11. Internal property storage design that allows for proper `Symbol` property keys integration.

IV. Evaluation

This chapter presents the evaluation of our implementation of the ES6 built-in libraries. In a nutshell, we evaluate our implementation by testing it against Test262, the ECMAScript official test suite. Even though we focus the evaluation on the `Symbol` built-in library discussed in Section III, we present our testing results for all built-in libraries.

A. Test262

The evaluation process of the ECMARef6 reference interpreter is straightforward given the existence of Test262 [6]. Since our main goal is to conform to the ECMAScript standard and the test suite's purpose is to test the conformity of an implementation to the standard, we can evaluate the extent and correctness of our implementation quantitatively, by checking the number of passing tests for each implemented library and contrasting that with the total number of tests for that library.

The tests that compose the Test262 test suite are JavaScript files with a set of instructions and the necessary assertions to verify that the state produced by the execution matched the tests' expectations.

Figure 12 is an example of a test file, which we can see is just a JavaScript file. It is also important to note the three distinct sections of the test file: the *copyright* section which has information related to the authors of the test; the *metadata* section where some important characteristics of the test are defined; and the *body* section where the JavaScript code resides. When it comes to ES6 tests, the metatadata section has a key-value structure with following keys:

- `es6id`: this value refers to the section of the standard targeted by the test;

- `description`: a succinct description of the feature being tested;
- `info`: information about the specific pseudo-code instruction of the standard that captures the core functionality being tested;
- `includes`: a collection of JavaScript files that need to be evaluated before executing the test code;
- `features`: the features of the standard that are being tested.

```
1  // Copyright (C) 2016 the V8 project authors. All
       rights reserved.
2  // This code is governed by the BSD license found in
       the LICENSE file.
3  /*---
4  es6id: 22.2.2.2
5  description: >
6    "of" cannot be invoked as a function
7  info: |
8    22.2.2.2 %TypedArray%.of ( ...items )
9
10   ...
11   3. Let C be the this value.
12   4. If IsConstructor(C) is false, throw a TypeError
         exception.
13   ...
14 includes: [testTypedArray.js]
15 features: [TypedArray]
16 ---*/
17
18 var of = TypedArray.of;
19
20 assert.throws(TypeError, function() {
21   of();
22 });
```

Fig. 12. Example of a test file of the Test262 suite.

For instance, the test given in Figure 12 tests the behavior defined in Section 22.2.2.2 of the standard, which defines the `%TypedArray%.of` function. More concretely, the instruction in the 4th line is supposed to throw a `TypeError` exception since ""of" cannot be invoked as a function". On line 20, there is an `assert` object and a call to its `throws` method neither of which is defined in the standard and therefore should not be accessible since they are not defined before line 20. This means that they are defined somewhere else, more concretely, the Test262 *harness*. The harness is a collection of JavaScript files composed of function and variable definitions, some of which must be executed before the test, more specifically the ones mentioned in the `includes` value of the `metadata` section.

1) Test Selection: Although, we use the Test262 suite to perform our evaluation, some of its tests are meant to target features of the newer versions of ECMAScript. Naturally, some tests targeting ES12, the newest version of the standard, are expected to fail when run against ECMARef6. Including these tests in our evaluation would pollute our results and prevent us from getting a clear idea of how well our implementation performed relative to the version of the standard that we target. This means that in order to get a correct assessment of the state of our implementation we need to filter out a portion of these tests.

Selecting tests is not trivial because not all of them come annotated with a flag that indicates their version. Up to 2016, tests included a flag indicating whether they target version 5 (`es5id`) or version 6 (`es6id`). These flags have deprecated. Hence, if a test comes with either the `es5id` or the `es6id` flags, then it should be included. However, the opposite does not hold. There might be tests without these flags that should also be included. The test filtering problem is a highly complex problem that cannot be systematically addressed in the context of this thesis. Our solution was to filter the unlabeled tests manually. More concretely, our selection methodology for unlabeled tests was simple: when testing our implementation, if the cause of failure of a test was the absence of features introduced in later versions of the standard, the test was discarded. Using this methodology, 81 out of 92 possible `Symbol` tests were selected.

B. Evaluation Pipeline

The execution pipeline of JavaScript files in ECMA-SL has 4 steps:

1) The abstract syntax tree (AST) of the JavaScript program is built, via the JS2ECMA-SL tool. The output produced is an ECMA-SL file, called **ast.esl**, containing a single function called **buildAST**, that will build the JavaScript program's AST in the ECMA-SL heap.
2) The **ES6_interpreter.esl** file that contains the reference interpreter is imported into the **ast.esl** file;
3) The next step is to compile this file into the Core version of ECMA-SL using the ECMA-SL2CoreECMA-SL tool, generating a CoreECMA-SL file, **core.cesl**.
4) The final step is to use the CoreECMA-SL interpreter, built with the OCaml language, to run the **core.cesl** file.

The test execution pipeline is similar to the one explained above except that the JavaScript file would be the test that we intend to run and the output of the `CoreECMA-SL` interpreter needs to be evaluated to determine the outcome of the test. There are 4 possible test outcomes which are determined by the exit code of the `CoreECMA-SL` interpreter:

- 0: **Ok** - the test passed;
- 1: **Fail** - the test failed because some of the assertions made in the test file were not true;
- 2: **Error** - the test failed because there was an internal error in the ECMARef6 interpreter, such as accessing a property of an undefined value or calling an internal function that does not exist or with the incorrect number of arguments;
- 3: **Unsupported** - the test failed because it requires some feature which is currently not implemented, such as one of the built-in libraries or a language feature like template literals, and so is expected to fail.

As discussed in Section IV-A, the Test262 harness must be executed before the body of the test so that the auxiliary testing functions can be defined. To fulfill this requirement we simply prepend the harness's code to the test's code.

Putting it all together, our testing pipeline, illustrated in Figure 13, starts with the concatenation of the harness and

test to be executed. That JavaScript file will then be parsed and compiled to ECMA-SL so that the ECMARef6 interpreter can be imported into it, creating the out.esl file of the diagram. This file is then compiled to CoreECMA-SL using the ECMA-SL2CoreECMA-SL tool, so that the code can be evaluated by the `CoreECMA-SL` interpreter and test's outcome determined.

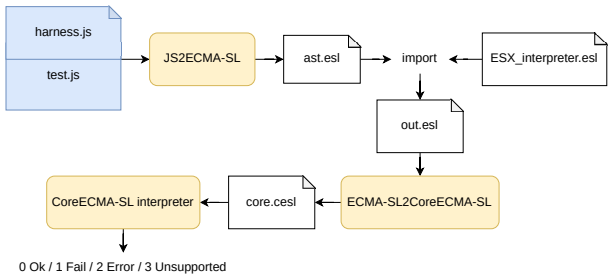

Fig. 13. Test execution pipeline.

C. Results

Considering that the measure we are using to evaluate the results obtained during this thesis is the conformity to the ECMAScript standard using the Test262 test suite, the various test outcomes described in Section IV-B can be considered irrelevant, as fail, error and unsupported test results all represent the same result in the evaluation context, that the interpreter does not conform.

Grouping all the negative test results and looking specifically at the tests that are related to the built-in libraries, we can do a proper assessment of the work performed during this thesis. Consider Table III that summarizes the results across all the built-in libraries of the ECMAScript standard. Here we can observe that although we employ strategies to guarantee the reference interpreter to the specification, there are still some errors present.

V. Conclusions

In this thesis we have worked in the context of the ECMA-SL project with the goal of extending its more up-to-date reference interpreter (ECMARef6) with support for the built-in libraries of the 6th version of the ECMAScript standard. This was done using the ECMA-SL language which was specifically designed to be similar or identical to the standard's pseudo-code. This similarity allowed us to use a line-by-line strategy, where we matched each pseudo-code instruction of the specification with an ECMA-SL statement in the implementation. This strategy gives us confidence that our implementation can be used as a reference for the ECMAScript standard, and serve as its own executable specification.

As the complexity of the ECMAScript standard increases, it becomes progressively more relevant the existence of a complete reference interpreter that can be used to reason about other implementations and as a testing mechanism. We believe ECMARef6 to be the reference interpreter with the most complete implementation of the built-in libraries of the standard, making its use as a testing oracle possible, since the built-in libraries are a large part of the ECMAScript language and most ECMAScript programs use them during their execution.

In the journey to attaining we had to extend the ECMA-SL language itself and change fundamental design decisions related to the core representation of ECMAScript objects. More concretely, we needed to extend the ECMA-SL language with two types, byte and array, and operators to create and manipulate them. This was necessary as with the previous version of the ECMA-SL language, it was impossible to represent the `Data Block` type introduced with the `ArrayBuffer` library. In the implementation of the `Symbol` library, we had to update the internal model of ECMAScript objects to support the use of `Symbol` values as property keys.

a) Future Work: As a continuation of the work done in this thesis, future work could be done to complete the implementation of the ECMARef6 reference interpreter. Core language functionality is not yet implemented, such as execution context switching, which is required for the implementation of the `Generator` and `GeneratorFunction` built-in libraries. Having a complete reference interpreter opens up a great many number of other possibilities for future work. The following are examples of possible projects:

1) A tool capable of generating the HTML document corresponding to the specification using the reference interpreter's code;
2) A tool with the inverse function can be done. It would use the specification to generate a reference implementation. A tool that was actually able to perform this function at a high level would significantly reduce the implementation time of future reference implementations, such as ECMARef7, ECMARef8, etc;
3) With a complete reference interpreter one could employ automatic test generation techniques to automatically create a conformance test suite, that could potentially complement Test262 as the official test suite of the standard.

Even without an automatic translation tool to generate reference implementations, future work could still be done to keep updating the current reference interpreters to the more recent versions of the standard, even if by hand.

References

[1] Coq - interactive formal proof management system. https://coq.inria.fr. Accessed on 2022-10-30.
[2] Ecmascript® 2015 language specification, 6th edition / june 2015. https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf. Accessed on 2022-10-31.
[3] Ecmascript® language specification, 5.1 edition / june 2011. http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf. Accessed on 2020-06-07.
[4] K - rewrite-based executable semantic framework. https://kframework.org/. Accessed on 2022-10-30.
[5] Ocaml - general-purpose, multi-paradigm programming language. https://ocaml.org/. Accessed on 2020-06-07.
[6] Test262 - official ecmascript conformance test suite. https://github.com/tc39/test262. Accessed on 2022-10-30.
[7] V8 - google's open source high-performance javascript and webassembly engine, written in c++. https://v8.dev. Accessed on 2022-10-30.

Section	# of tests	Passed	Failed	Passed Percentage
19.4 (Symbol)	81	68	13	83.95%
19.1 (Object)	2942	2926	16	99.46%
19.2 (Function)	399	378	21	94.74%
19.3 (Boolean)	51	51	0	100.00%
19.5 (Error)	41	41	0	100.00%
20.1 (Number)	348	303	45	87.07%
20.2 (Math)	341	337	4	98.83%
20.3 (Date)	750	741	9	98.80%
21.1 (String)	1014	973	41	95.96%
21.2 (RegExp)	1410	885	525	62.77%
22.1 (Array)	2701	2675	26	99.04%
22.2 (TypedArray)	959	942	17	98.22%
23.1 (Map)	156	154	2	98.72%
23.2 (Set)	197	196	1	99.49%
23.3 (WeakMap)	93	91	2	97.85%
23.4 (WeakSet)	79	78	1	98.73%
24.1 (ArrayBuffer)	79	77	2	97.46%
24.2 (DataView)	327	324	3	99.08%
24.3 (JSON)	150	138	12	92.00%
25.1 (Iteration)	4	4	0	100.00%
25.2 (GeneratorFunction)				Not Implemented
25.3 (Generator)				Not Implemented
25.4 (Promise)	384	375	9	97.66%
26.1 (Reflect)	152	149	3	98.02%
26.2 (Proxy)	259	255	4	96.53%
26.3 (Module Namespace)				Not Implemented
Total	13253	12457	793	93.99%

TABLE III
TEST RESULTS OF THE BUILT-IN LIBRARIES.

[8] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. volume 49, pages 87–100, 01 2014.

[9] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. Jsexplain: A double debugger for javascript. pages 691–699, 04 2018.

[10] Discord — your place to talk and hang out. Accessed on 2022-01-14.

[11] Electron — build cross-platform desktop apps with javascript, html, and css. Accessed on 2022-01-14.

[12] Jose Fragoso Santos, Petar Maksimović, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. Javert: Javascript verification toolchain. *Proceedings of the ACM on Programming Languages*, 2:1–33, 12 2017.

[13] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a program logic for javascript. volume 47, pages 31–44, 01 2012.

[14] D. Gonçalves. A reference implementation of ecmascript built-in objects. Master's thesis, Instituto Superior Técnico, October 2021.

[15] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. pages 126–150, 06 2010.

[16] L. Loureiro. Ecma-sl - a platform for specifying and running the ecmascript standard. Master's thesis, Instituto Superior Técnico, July 2021.

[17] Sergio Maffeis, John Mitchell, and Ankur Taly. An operational semantics for javascript. pages 307–325, 12 2008.

[18] Node.js. Accessed on 2022-01-14.

[19] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. *ACM SIGPLAN Notices*, 50:346–356, 06 2015.

[20] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. *SIGPLAN Not.*, 48(2):1–16, oct 2012.

[21] F. Quinaz. Precise information flow control for javascript. Master's thesis, Instituto Superior Técnico, July 2021.

[22] Tc39 - specifying javascript. Accessed on 2022-10-30.

[23] The tc39 process. Accessed on 2022-10-30.

[24] Visual studio code - code editing. redefined. Accessed on 2022-01-14.