TÉCNICO LISBOA

A Reference Implementation of ECMAScript Built-in Objects

David Manuel Sales Gonçalves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. Nuno Miguel Carvalho dos Santos

November 2021

Acknowledgments

This work would not have been possible without the numerous engineers that have preceded me and the core values and principles of the civilization that supported them in their creativity and endeavours. Special thanks to Marc Andreessen and Brendan Eich for the birth of ECMAScript, whose legacy this work aspires to continue.

I would like to express my gratitude and appreciation to my family for granting me the conditions to make this project a reality, and to my supervisor, Prof. Dr. José Fragoso Santos, for suggesting this topic to me and for his undying motivation and assistance throughout the course. I would also like to thank Prof. Dr. António Leitão for his review, feedback and enlightenment in what would become of this project and my colleague Luís Loureiro who I worked in close collaboration with for the larger project in which this one is integrated.

Abstract

ECMAScript (ES), commonly known as JavaScript, is one of the most important programming languages today because it is the *de facto* option when it comes to dynamic front-end web development. Throughout the years, ES has become an increasingly complex language, making it a difficult target for static analyses. This project is part of a wider project that aims to build a trustworthy reference interpreter for ES, which will, among other things, enable the development of precise static analysis tools for modern ES applications through the use of a novel intermediate language called ECMA-SL. This work focuses on implementing three built-in objects of ECMA-262 Edition 5.1: Array (15.4), RegExp (15.10), and JSON (15.12). It also implements a few methods of the String (15.5) built-in object and implements the Promise (25.4) built-in object of the ECMA-262 6th Edition. This implementation scrupulously follows the ES standard's pseudo-code line-by-line, thus ensuring that our interpreter is correct with respect to the standard and allowing us to regard the interpreter as the standard itself in the context of static analysis. To this end, we also extend the ECMA-SL execution engine with new programming constructs, including UTF-8 support. Furthermore, our reference implementation is thoroughly tested against Test262, the official ECMAScript test suite. Finally, in order to assist with the transition from the current HTML representation of the standard to ECMA-SL, we introduce `HTML2ECMA-SL`, a tool that aims to generate the ECMA-SL code for any given function described in the standard.

Keywords: ECMAScript, Specification Language, Reference Interpreters, Dynamic Languages, Test262, OCaml

Resumo

ECMAScript (ES), vulgarmente conhecida como JavaScript, é uma das linguagens mais importantes de hoje porque é a opção *de facto* para desenvolvimento *web front-end*. Ao longo dos anos, ES tornou-se uma linguagem cada vez mais complexa, tornando-se um alvo difícil para análises estáticas. Este projeto faz parte de um projeto mais amplo que visa construir um interpretador de referência fiável para ES, que permitirá, entre outras coisas, o desenvolvimento de ferramentas de análise estática precisas para aplicações ES modernas através do uso de uma nova linguagem intermédia chamada ECMA-SL. Este trabalho foca-se na implementação de três bibliotecas *built-in* da edição 5.1 do *standard* de ES: Array (15.4), RegExp (15.10), e JSON (15.12). Também implementa alguns métodos da biblioteca String (15.5) e implementa a biblioteca Promise (25.4) da 6ª edição do *standard*. Esta implementação segue escrupulosamente o pseudo-código do *standard* linha por linha, assegurando assim que o nosso interpretador está correto com respeito ao *standard* e permitindo-nos considerar o interpretador como o próprio *standard* no contexto da análise estática. Para este fim, também estendemos o motor de execução da ECMA-SL com novas construções de programação, incluindo suporte para UTF-8. Além disso, a nossa implementação de referência é testada na totalidade com a Test262, a *suite* de testes oficial para ECMAScript. Finalmente, a fim de auxiliar na transição da representação HTML atual do *standard* de ES para ECMA-SL, apresentamos `HTML2ECMA-SL`, uma ferramenta que visa gerar o código ECMA-SL para qualquer função descrita no *standard*.

Palavras-Chave: ECMAScript, Linguagem de especificação, Interpretadores de referência, Linguagens dinâmicas, Test262, OCaml

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 1

AS ActionScript. 15

ASCII American Standard Code for Information Interchange. 29–31

AST Abstract Syntax Tree. 25, 43, 44, 61

BMP Basic Multilingual Plane. 30, 32, 33, 42

BOM Byte Order Mark. 30

DOM Document Object Model. 1, 23

DRY Don't Repeat Yourself. 17

ECMA European Computer Manufacturers Association. 1

ECMA-262 Standard of the ECMAScript programming language. xi, 2–5, 7, 8, 11–15, 17, 19–26, 31, 35, 36, 40, 46, 47, 50, 55–57, 59–63

ECMA-404 Standard of the JSON Data Interchange Syntax. 14, 34, 48

ECMA-SL ECMAScript Specification Language. ix, xi, 1–4, 9, 23–28, 30–33, 35–37, 40–50, 55–57, 59–63, 69

ES ECMAScript. xi, 1–3, 5–15, 17, 18, 22–26, 28, 31–33, 35, 36, 39, 47, 48, 55, 56, 59–61, 63

ES12 ECMAScript 12th Edition. 59

ES3 ECMAScript 3rd Edition. 21, 22

ES5 ECMAScript 5.1 Edition. ix, xi, 3–6, 9–12, 14, 17, 20, 22, 24, 25, 31, 33–37, 39–50, 55, 59–61, 63, 71–73

ES6 ECMAScript 6th Edition. ix, xi, 2–6, 14, 17, 18, 20, 23, 25, 32, 33, 35, 44, 48–50, 53, 55, 57, 59, 60, 62, 63

ES8 ECMAScript 8th Edition. 2

HTML HyperText Markup Language. 1, 3, 13, 14, 26, 47, 55–57, 59, 60

IL Intermediate Language. 2, 25

JS JavaScript. 1, 21–23, 25, 26, 29, 30, 56

JSON JavaScript Object Notation. ix, xi, 3–5, 14–16, 20, 23, 24, 33–35, 46–48, 61, 63, 71

LOC Lines of Code. 37, 41, 45, 47, 50, 57, 63

PDF Portable Document Format. 55

RegExp Regular Expression. ix, xi, 3–5, 9, 12–14, 20, 22–24, 33, 35, 41, 43–45, 57, 61, 63, 73

Test262 Official ECMAScript Conformance Test Suite. ix, xi, 2, 3, 20–23, 31, 33, 42, 43, 50, 59–63, 71–73

TS TypeScript. 56, 57

UCS Universal Character Set. 29

UCS-2 Universal Character Set with a 16-bit encoding. 29–31

UCS-4 Universal Character Set with a 32-bit encoding. 29, 30

UI User Interface. 23

URL Uniform Resource Locator. 56

UTF Unicode Transformation Format. 33

UTF-16 Unicode Transformation Format — 16-bit. 10, 29–33, 42

UTF-32 Unicode Transformation Format — 32-bit. 29–31

UTF-8 Unicode Transformation Format — 8-bit. ix, 3, 29–33, 61

Chapter 1

Introduction

In this chapter, we motivate the reader by highlighting the importance that JavaScript has in the world today (1.1). Then, we raise awareness to the complexity of JavaScript's standard document (1.2), which is one of the problems that this work attempts to address. We then introduce ECMA-SL, a novel intermediate language that aspires to solve many of the problems with the specification and uses of JavaScript in modern web applications (1.3). Finally, we give an overview of the problems addressed by this work (1.4) and the structure of this document (1.5).

1.1 Why JavaScript?

JavaScript (JS) is one of the most important programming languages today because it is the *de facto* option when it comes to dynamic front-end web development. Whilst there are viable alternatives (such as TypeScript, CoffeeScript, and so on), the fact is that these end up being transpiled into JavaScript. JavaScript is used by over 95% of websites[1], is the most active language on GitHub[2] and the second most active on StackOverflow[3]. Throughout the years, a rich ecosystem was built on top of JS, from transpilers to frameworks, such that, for a web developer, it is almost impossible to avoid working with JS in some way or another.

JavaScript was envisioned by Marc Andreessen, founder of Netscape Communications, and created by Brendan Eich in September 1995, who completed the first version in only ten days in order to accommodate the Navigator 2.0 Beta release schedule [1]. Originally, it had the code name Mocha but was launched as LiveScript and soon after renamed to JavaScript for marketing purposes, which was possible due to Netscape's collaboration with Sun Microsystems, who had been responsible for the creation of the Java programming language. At the same time, Microsoft was working on its own web browser, Internet Explorer, and after witnessing the success of JS on the Netscape Navigator it had no choice but to come up with its own implementation of JS, which they called JScript in order to avoid trademark issues with Sun Microsystems. To prevent competition between scripts, Netscape submitted JS to the European Computer Manufacturers Association (ECMA) in November 1996, as the starting point for a standard specification that all browser vendors could conform to. This led to the official release of the first ECMAScript (ES) language specification in June 1997. It is worth mentioning that Web APIs such as the HTML DOM API are not a part of ES and have their own standards.

[1] Usage statistics of JavaScript as client-side programming language on websites, 31 October 2021, W3Techs.com - `https://w3techs.com/technologies/details/cp-javascript`

[2] Github most active programming languages based on pull requests, 31 October 2021 - `https://madnight.github.io/githut`

[3] Stack Overflow Trends over time based on use of their tags, 31 October 2021 - `https://insights.stackoverflow.com/trends`

JavaScript, whose official name is now ECMAScript, is an ever-evolving programming language that keeps backward compatibility with its earlier versions. It is currently in its 12th edition. However, a large portion of JavaScript's current features were introduced with the release of ECMAScript 6 (ECMAScript 2015), which has added new syntax for writing more complex applications, among many other features that would define the next era of JavaScript.

1.2 ECMA-262's Complexity

ECMA-262 [2] is the specification of the ECMAScript (ES) programming language, standardized by Ecma International. Since 2015, the specification underwent yearly updates. Figure 1.1 shows how complex the standard has become, with the largest updates being introduced in ES6 and ES8.

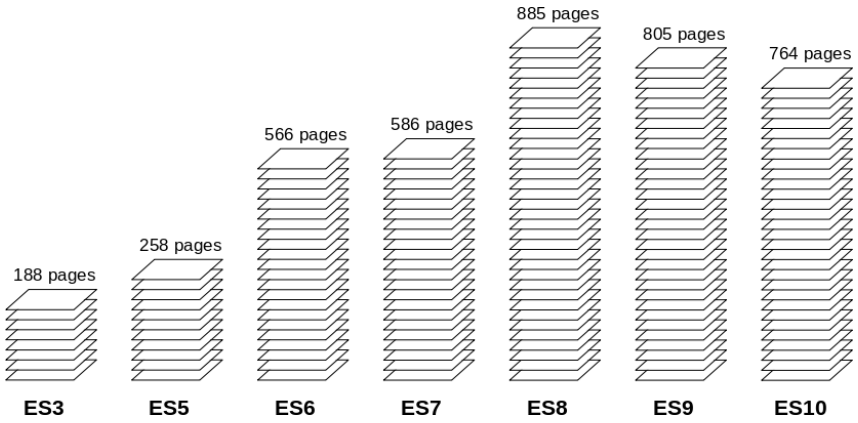


Figure 1.1: The evolution on the number of pages of the ES standard official document.

Given the current size and complexity of ECMA-262, introducing new features to the language is a complex and error-prone process. For instance, it must be guaranteed that any change does not break the behaviour of previous features and that it is compatible with the internal invariants maintained by the semantics of the language. For this reason, the ECMAScript committee created Test262 [3], the official ECMAScript Conformance Test Suite, which is known to have significant coverage issues. Even with extensive testing, one might fail to cover an edge case of the language. In contrast to testing, formal methods offer strong correctness guarantees. But, in order to be able to apply formal methods to the ECMA-262 standard, one must first have a formal model of it.

1.3 ECMA-SL Project

The ECMAScript Specification Language (ECMA-SL) is a dedicated intermediate language (IL) for ES analysis and specification that aims to mitigate the problems caused by the complexity of ECMA-262. One of the goals of the ECMA-SL project is to have a reference interpreter for ES that is tightly linked to the text of ECMA-262, allowing us to regard the interpreter as ECMA-262 itself in the context of formal analysis. We can also generate a textual version of ECMA-262 from the ECMA-SL code, making ECMA-SL a viable alternative for specifying the ES language. As we can test the ECMA-SL interpreter against Test262, we can: make sure that any changes are backward compatible with previous versions; apply test generation techniques [4] to automatically obtain tests for newly introduced features; and measure the coverage of Test262. The ECMA-SL project is currently in the development stage and ships a fully functional reference interpreter for the ECMA-262 Edition 5.1 [5], henceforth referred to as `ECMARef5`, in part contributed to by this work.

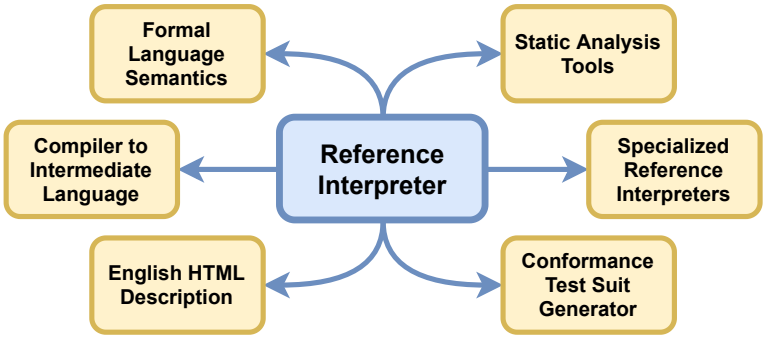


Figure 1.2: Tools that can be created from an ES reference interpreter.

1.4 Problem Statement

In this work, we extend `ECMARef5` in order to include the ES5 Array object, the ES5 RegExp object, and the ES5 JSON object, which were yet to be implemented with regards to the pseudo-code of the standard. We also implement a few methods of the ES5 String object, that depend on the ES5 RegExp object and Unicode, and start implementing the ECMA-SL reference interpreter for the ECMA-262 6th Edition, henceforth referred to as `ECMARef6`, by implementing the ES6 Promise object.

The ECMA-SL execution engine is implemented in the OCaml [6] programming language and includes a parser based on *Menhir* [7]. In order to implement the aforementioned built-in objects, we extend the ECMA-SL execution engine with new programming constructs and UTF-8 support. As part of this work, we also create a tool for converting the HTML representation of the ES6 standard directly to ECMA-SL code, which we call `HTML2ECMA-SL`. So far we can only guarantee full support of the ES6 Promise object for `HTML2ECMA-SL`. `HTML2ECMA-SL` has a number of benefits:

1. *It guarantees consistency*: The ECMA-SL instructions generated by this tool are kept consistent across functions. Sometimes, there can be multiple ways to implement a given pseudo-code instruction, and, by adhering to one single way, the inverse process of converting the ECMA-SL code to a textual representation of the standard is facilitated. Moreover, different programmers may have different styles of coding (e.g. placement of braces, spaces around operators, function names, etc.). `HTML2ECMA-SL` helps with style standardization.

2. *It avoids errors*: Manual tasks are known for being error-prone. It is easy for the programmer to make mistakes which can take a long time to debug and detect.

3. *It speeds up the implementation*: Whilst there is an initial overload on implementing this tool alongside the ECMA-SL reference implementation, as more statements and expressions are supported by the tool the easier and quicker it becomes to implement more functions from the ECMA-262's pseudo-code along the line.

We thoroughly evaluate the three main modules of this work: The extensions to the ECMA-SL execution engine, the built-in objects added to the ECMA-SL reference interpreter, and the `HTML2ECMA-SL` conversion tool. The string operators added to the ECMA-SL execution engine concerning UTF-8 were partially tested using the *OUnit* [8] testing framework and all the extensions to the ECMA-SL execution engine were tested together with the ECMA-SL reference interpreter against Test262. Out of a total of 3,440 applicable tests from Test262, we pass 99.8%, thus guaranteeing that our reference implementation is correct with respect to the ECMA-262 standard. `HTML2ECMA-SL` is evaluated using the Jest [9] testing framework, by comparing each generated ECMA-SL function's code to the code that we deem to be correct. `HTML2ECMA-SL` passes 51 out of 51 unit tests and has a test coverage of 97%.

1.5 Thesis structure

This document is organized as follows: Chapter 2 introduces ECMA-262 and the built-in objects that this work focuses on, explaining their features, methods and evolution from edition 5.1 [5] to the 6th edition [10]. Chapter 3 presents several research works with similarities to our own. Chapter 4 introduces ECMA-SL and our extensions to this language. Chapter 5 presents our implementation of the built-in objects that this work focuses on, starting with the ES5 Array object (5.1), followed by part of the ES5 String object (5.2), the ES5 RegExp object (5.3), the ES5 JSON object (5.4), and the ES6 Promise object (5.5); for each of these we give an overview of their internal representation, internal functions (or abstract operations), reference implementation, implementation-dependent methods (if any), and auxiliary functions. Chapter 6 presents `HTML2ECMA-SL`. Chapter 7 presents the evaluation of the main outcomes of this work: Reference implementations (7.1) and `HTML2ECMA-SL` (7.2). Finally, Chapter Chapter 8 draws some conclusions about our work and points out some future research directions.

Chapter 2

Background

In this chapter, we give an overview of the ECMAScript standard (2.1) and explain each of the built-in objects and methods that this work focuses on: the ES5 Array object (2.2), part of the ES5 String object (2.3), the ES5 RegExp object (2.4), the ES5 JSON object (2.5), and the ES6 Promise object (2.6). We also mention some of the incompatibilities of ES6 with the prior editions (2.7).

2.1 ES Standard

Edition 5.1 [5] of ECMA-262 defines the ECMAScript programming language for version 5.1, hereafter referred to as ES5, and the 6th edition [2] defines version 6 of the language, hereafter referred to as ES6. ECMA-262 defines the types, values, objects, properties, functions, program syntax and semantics that should exist in an ES language implementation. Note that ECMA-262 allows an implementation of the language to provide additional types, values, objects, properties, and functions [1].

ECMAScript defines a collection of built-in objects whose specification comprises almost half of the 6th edition [2] of the ECMA-262 standard (253 out of 545 pages). Built-in objects provide the essential functionality of the language. The built-in objects present in ES5 are the *Global* object, the *Object* object, the *Function* object, the Array object, the String object, the *Boolean* object, the *Number* object, the *Math* object, the *Date* object, the RegExp object, the JSON object, and the following Error objects: *Error*, *EvalError*, *RangeError*, *ReferenceError*, *SyntaxError*, *TypeError*, and *URIError*.

ES6 is mostly compatible with the previous versions, with a few exceptions [2], so it contains the same built-in objects that are present in ES5. However, it introduces the following new built-in objects: *ArrayBuffer*, *DataView*, *Float32Array*, *Float64Array*, *Generator*, *GeneratorFunction*, *Int8Array*, *Int16Array*, *Int32Array*, *Map*, *Proxy*, Promise, *Reflect*, *Set*, *Symbol*, *Uint8Array*, *Uint8ClampedArray*, *Uint16Array*, *Uint32Array*, *WeakMap* and *WeakSet*. ES6 also introduces the following iteration interfaces: *Iterable*, *Iterator*, and *IteratorResult*.

Figure 2.1 illustrates the ECMAScript language 6th edition's built-in objects. Of the several specified built-in objects, this work focuses on the ES5 Array object, part of the ES5 String object, the ES5 RegExp object, the ES5 JSON object, and the ES6 Promise object.

[1] Standard ECMA-262's 6th Edition, Chapter 2, Paragraph 4.
[2] Standard ECMA-262's 6th Edition, Annex D and Annex E.

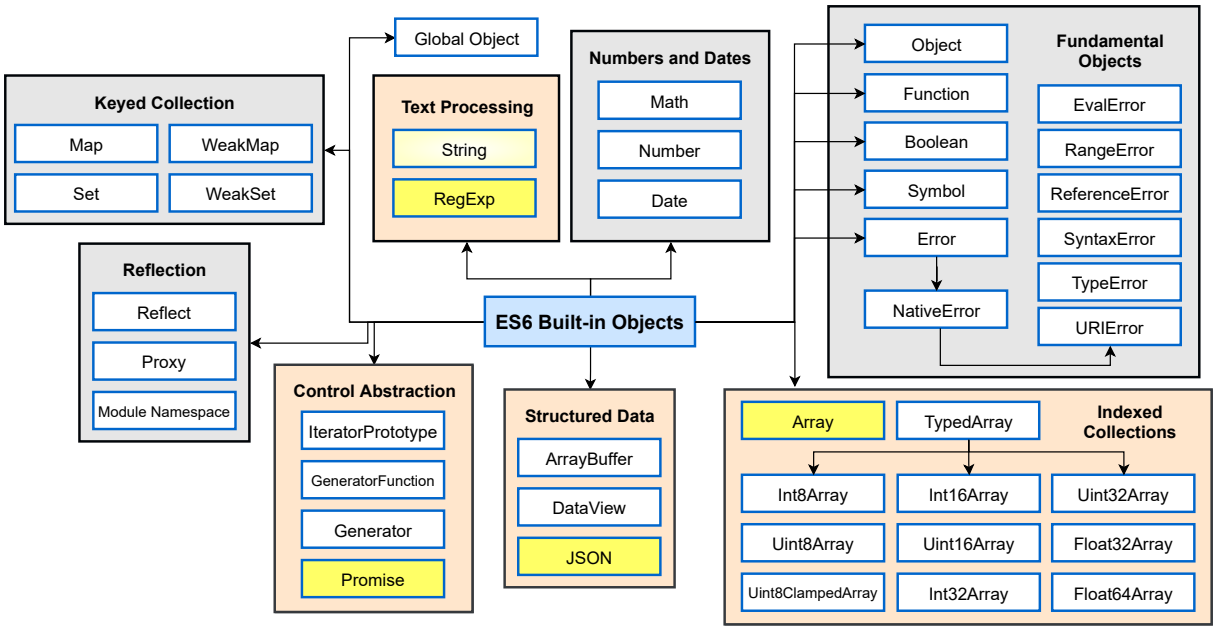


Figure 2.1: A graphical overview of the ECMAScript language 6th edition's built-in objects.

2.2 ES5 Array Object

Indexed Collections include the Array object, also present in ES5, as well as nine different variants of typed arrays, provided by the TypedArray library, whose elements have a specific numeric data representation and have been introduced in ES6. This work focuses only on the ES5 Array object. Figure 2.2 illustrates the internal representation of a typical ES5 Array object according to the standard.

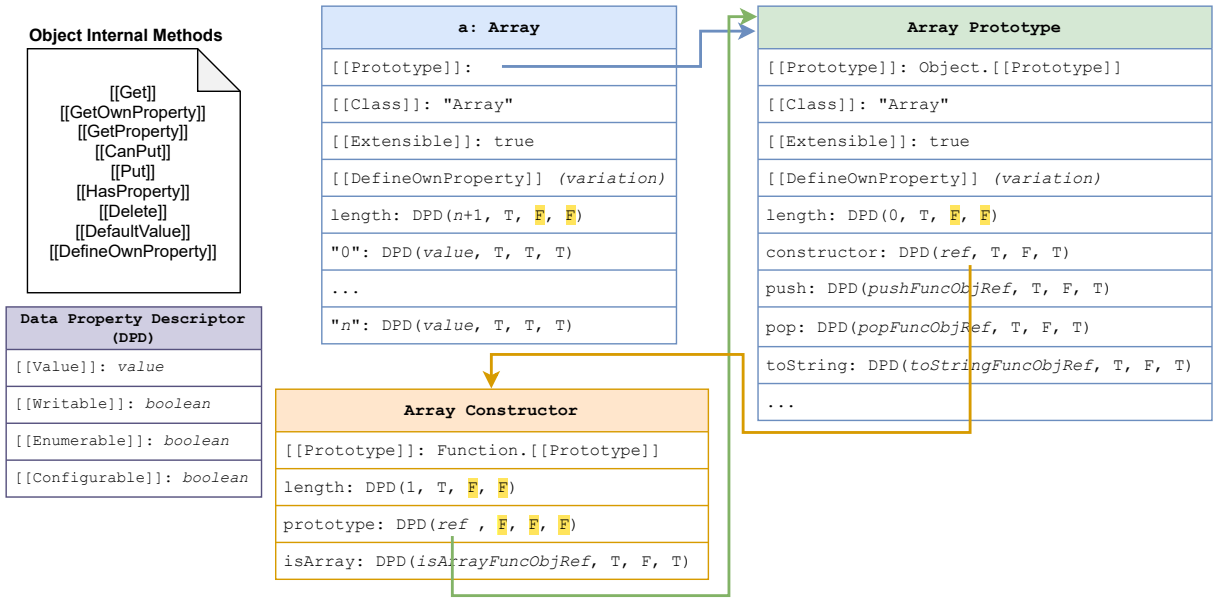


Figure 2.2: ES5 Array Object Graph.

In order to understand this figure, we first need to introduce some basic concepts related to the representation of objects and arrays in ES.

First, we observe that ES arrays are stored in memory as objects. As all ES objects, an Array object has two types of properties: internal properties and named properties. Internal properties, which we

represent between double square brackets, store the meta-data and internal algorithms associated with the object and cannot be directly accessed or modified by an ES program; they include the properties:

- `[[Class]]` representing the type of the object in the form of a string;
- `[[Extensible]]` storing a boolean that determines whether or not it is possible to add new named properties to the object;
- `[[Prototype]]` representing the internal prototype of the object (used to implement prototype-based inheritance [11]). Its value is either a pointer to an object, also referred to as an object location, or `null`.

Named properties are the properties explicitly set by the program, either through built-in functions or assignment expressions. They are represented by *Property Descriptors*. A property descriptor is a record with specific attributes representing both the property value and meta-information about the property. There are three types of property descriptors, but here we focus only on data property descriptors, which store the following four attributes:

- `[[Value]]` holds the actual property value;
- `[[Writable]]` determines whether the property value may or may not be modified;
- `[[Enumerable]]` determines whether the property is to be visible by operations that iterate over the properties of the object, such as `for-in` enumerations;
- `[[Configurable]]` determines whether the property can be deleted, have its attributes changed (other than `[[Value]]`), or if it can be transformed into an accessor property descriptor.

When it comes to Array objects, ECMA-262 distinguishes two types of named properties: *indexed properties*, corresponding to array indexes, and *non-indexed properties*. Importantly, just like any other property, indexed properties are also associated with property descriptors. By default, indexed properties are writable, enumerable, and configurable. Given that they are writable and configurable, all of their attributes can be changed at runtime.

Besides indexed properties, all Array objects have a distinguished property `length`, representing the length of the array. The length of an array corresponds to the value of its highest index incremented by one. In Figure 2.2, we can see that the attributes `[[Enumerable]]` and `[[Configurable]]` of the `length` property descriptor are set to false. `[[Enumerable]]` is false because the property `length` is not supposed to be visible to operations that iterate over the properties of an object, such as `for-in` enumerations. `[[Configurable]]` is false because the property `length` cannot be deleted nor have its attributes changed (other than `[[Value]]`).

In ES, all objects have an internal prototype from which they can inherit properties. As arrays are represented in memory as objects, Array objects also have an internal prototype—the Array Prototype object—which stores the methods shared by all Array objects (e.g. push, pop, etc.). Given that the Array Prototype object is itself an object, it also has an internal prototype—the Object Prototype object—which stores the methods shared by all ES objects, regardless of their type (e.g. toString, hasOwnProperty, etc.). This realises prototype-based inheritance [11].

ES functions are also represented in memory as objects. Hence, the Array constructor is itself an object. Function objects have a distinguished named property, `prototype`, where they store the object to be used as the internal prototype of the objects constructed with that function. Hence, the property `prototype` of the Array constructor Function object points to the Array Prototype object, which, as previously mentioned, is the internal prototype of all Array objects. In the same fashion, the Array

Prototype object (as all prototype objects) stores a reference to the constructor function of the objects that inherit from it in its named property `constructor`, which, in this case, is the Array constructor Function object. This effectively creates a circular dependency between the Array constructor object and the Array Prototype object, allowing us to write silly code like the one presented in Listing 2.1.

```
1 var arr = new Array.prototype.constructor.prototype.constructor();
2 // Equivalent to: var arr = new Array();
```

Listing 2.1: A silly Array object instantiation.

It is worth noting that the Array prototype object is itself an array and that all the built-in methods that can be called on an Array object are inherited from the Array prototype object. Another important characteristic of Array objects that is worth mentioning is that these objects are exotic objects (the concept of exotic object is formally introduced in Definition 2.2.1).

Definition 2.2.1 (Exotic Object)**.** *An exotic object is an object that does not have the default behaviour for one or more of the essential internal methods that must be supported by all objects. Array objects, for instance, provide an alternative definition for the [[DefineOwnProperty]] internal method.*

In order to understand why this matters, we use Listing 2.2 to illustrate a practical example of a strange case with an Array object, which will promptly be explained ahead with the aid of the pseudo-code present in the ECMA-262's 5.1 Edition [5].

```
1 var arr = [1, 2, 3];
2 Object.defineProperty(arr, "66", {
3   value: 666,
4   writable: true,
5   enumerable: true,
6   configurable: false });
7 arr.length = 2;
8 arr.length; // ??
```

Listing 2.2: A strange case with the array length.

In order to understand how this example works, we will also consider the ES heap resulting from the execution of lines 1–6 of the example, illustrated in the Figure 2.3.

Data Property Descriptor (DPD)
[[Value]]: *value*
[[Writable]]: *boolean*
[[Enumerable]]: *boolean*
[[Configurable]]: *boolean*

a: Array	
[[Prototype]]: Array.prototype	length: DPD(67, T, F, F)
[[Class]]: "Array"	"0": DPD(1, T, T, T)
[[Extensible]]: true	"1": DPD(2, T, T, T)
[[DefineOwnProperty]] *(variation)*	"2": DPD(3, T, T, T)
...	"66": DPD(666, T, T, F)

Figure 2.3: Properties of `arr` after executing lines 2–6.

For general ES objects, the core behaviour of the property update operation is described in the internal method `DefineOwnProperty` (DOP), given in section 8.12.9 of ECMA-262's 5.1 Edition [5]. Array objects, however, have their own `DefineOwnProperty` internal method (DOP-A), given in section 15.4.5.1 of ECMA-262's 5.1 Edition [5]. Among other things, DOP-A describes the behaviour of the ES semantics when trying to update the property `length` of an Array object. Figure 2.4 shows the exact snippet of this internal method that captures this behaviour. In line 3 of the figure, DOP-A checks if the property being

modified is the `length` property. Should this be the case, DOP-A iterates on all indexed properties of the array whose indexes are greater than or equal to the new length, in case the new length is smaller than the old length (line 3.l). For each iteration, the method first decrements the value of the `oldLen` variable (line 3.l.i) and tries to delete the current indexed property (line 3.l.ii). Then, it checks whether or not the delete operation was successful (line 3.l.iii). If it is successful, the loop continues. If it is not successful, the loop terminates and the length of the array is set to the value `oldLen + 1` (lines 3.l.iii.1–3.l.iii.4). Notice that, should the delete operation not be successful, the loop terminates with "Reject", which throws a `TypeError` exception if the `Throw` parameter is true, and returns false otherwise.

15.4.5.1 [[DefineOwnProperty]] (P, Desc, Throw) # Ⓣ Ⓔ

Array objects use a variation of the [[DefineOwnProperty]] internal method used for other native ECMAScript objects (8.12.9). Assume A is an Array object, *Desc* is a Property Descriptor, and *Throw* is a Boolean flag. In the following algorithm, the term "Reject" means "If *Throw* is **true**, then throw a **TypeError** exception, otherwise return **false**." When the [[DefineOwnProperty]] internal method of A is called with property P, Property Descriptor *Desc*, and Boolean flag *Throw*, the following steps are taken:

(...)

3. If P is `"length"`, then
   (...)
   l. While *newLen* < *oldLen* repeat,
      i. Set *oldLen* to *oldLen* – 1.
      ii. Let *deleteSucceeded* be the result of calling the [[Delete]] internal method of A passing ToString(*oldLen*) and **false** as arguments.
      iii. If *deleteSucceeded* is **false**, then
         1. Set *newLenDesc*.[[Value] to *oldLen+1*.
         2. If *newWritable* is **false**, set *newLenDesc*.[[Writable] to **false**.
         3. Call the default [[DefineOwnProperty]] internal method (8.12.9) on A passing `"length"`, *newLenDesc*, and **false** as arguments.
         4. Reject.

(...)

Figure 2.4: ES5 Array's `[[DefineOwnProperty]]` resizing the length.

We are now at the position where we can describe the behaviour of the program given in Listing 2.2. The array in the example is said to be *sparse*, because the indexes between 2 and 66, exclusive, are undefined. What makes this example unorthodox is that we set index 66's `[[Configurable]]` attribute to false, making it impossible to delete it. When setting the `[[Value]]` of the length of the array to 2, as per line 7 of the example, the JS engine will attempt to delete all the array elements whose indexes are greater than or equal to 2. This operation will not succeed, however, for the index 66, because it is not configurable. Hence, after the execution of line 7, the length of the array will continue to be 67 and a `TypeError` exception will be thrown in strict mode [3].

2.3 ES5 String Object

Text Processing objects include the String built-in object and the RegExp built-in object. The ES5 String object had already been partially implemented in ECMA-SL when this work was started. However, some methods of the String object depend on the RegExp object, which was yet to be implemented as part of this work. Additionally, operators in ECMA-SL that deal with Unicode "characters", and which the String object depends on, were also implemented as part of this work. For this reason, it was decided that the conclusion of the ES5 String object was also to be included in this thesis.

[3] ECMAScript's strict mode was introduced with ES5 as a way to avoid common pitfalls, which are possible due to to the backward compatibility of ES with its earlier versions that contained poorly written algorithms.

Strings are traditionally sequences of characters, although they can often store any sequence of bytes. They can be typically found in any useful programming language and resemble arrays, in the sense that they store a sequence of values that can be indexed. In ES, the pre-eminent differences between the String object and the Array object, is that the latter can store a sequence of values of any type and is mutable, whilst the former can only store a sequence of 16-bit code units [4] and is immutable. Furthermore, the String object is designed for the purpose of storing, manipulating, and displaying human-readable data, hence its methods provide useful operations in that regard.

In ES, a string is a primitive value corresponding to a sequence of zero or more 16-bit unsigned integers, because strings in ES use the UTF-16 character encoding, which is explained in more detail in Section 4.2. In ES5, string literals are denoted by single or double quotes and can contain certain escape sequences, such as Unicode code points [5] represented in hexadecimal. A string primitive can be wrapped in a String object, when passed to the String constructor, even though ES automatically coerces string primitives to String objects when a method is invoked on them. The explicit usage of String objects in ES is usually discouraged, as string objects are compared by reference while string primitives are compared by value. Listing 2.3 illustrates this difference. String objects may be explicitly coerced to string primitives by calling the String constructor as a function (i.e. without using the `new` keyword).

```
1 var s1 = "1";
2 var s2 = "1";
3 s1 == s2 // true
4 var S1 = new String("2");
5 var S2 = new String("2");
6 S1 == S2 // false
7 String(S1) == String(S2) // true
```

Listing 2.3: Comparing string primitives and String objects.

The internal representation of a typical ES5 String object is illustrated in Figure 2.5. The string primitive is stored in the `[[PrimitiveValue]]` internal property of the String object.

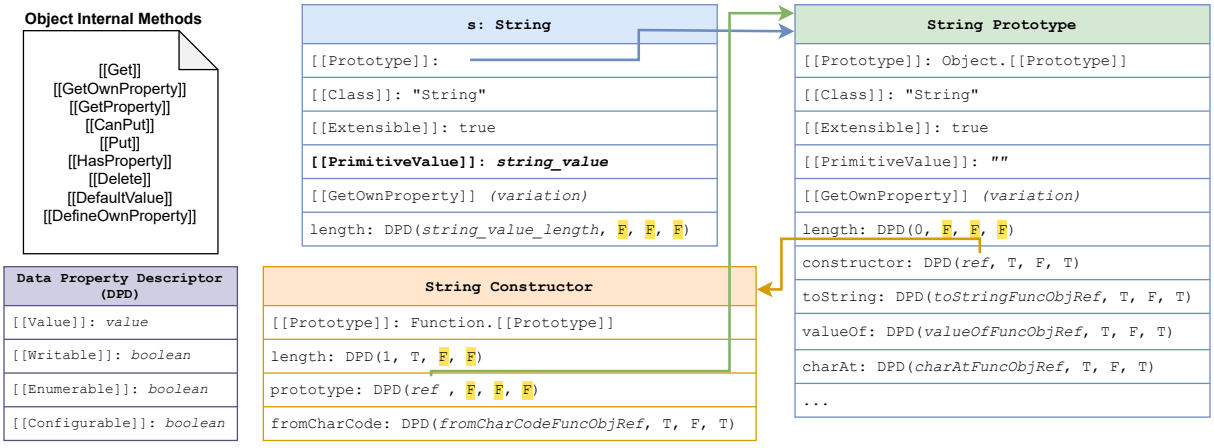


Figure 2.5: ES5 String Object Graph.

By having the length of a String object non-writable and non-configurable, we have an important hint about strings in ES—that they are immutable. In other words, the `[[PrimitiveValue]]` internal property of a String object cannot be changed, and a new string primitive or String object must be created if any

[4] A 16-bit code unit occupies 2 bytes. The concept of code units will be formally introduced in Section 4.2.
[5] The definition of code point is formally introduced in Section 4.2

alteration is to be made. In fact, none of the internal nor built-in methods of the String object change the internal property `[[PrimitiveValue]]`, returning a new string primitive instead, when applicable. Listing 2.4 demonstrates this.

```
1 var s = "This is a string.";
2 s.toUpperCase();
3 s // "This is a string."
4 s = s.toUpperCase();
5 s // "THIS IS A STRING."
```

Listing 2.4: A demonstration of string immutability.

Although ECMA-262 does not place any restrictions or requirements on how to store string primitives, one could take advantage of the immutability of String objects in order to save memory space, with string interning being the most obvious optimization to apply, where the ES engine only needs to keep one copy of each distinct string value handled by the program. String immutability can also have a few disadvantages in performance, however, especially when numerous changes are applied to a string (e.g. multiple string concatenations). In some programming languages where strings are immutable, such as Java and .NET, there are alternatives to immutable strings, such as the `StringBuilder` class, where strings are mutable and perform drastically better than the immutable counterpart in the aforementioned case. The ES standard does not include any mutable string alternative, but the performance drawback of string immutability can be tackled through the use of the rope data structure [12].

String objects can be accessed as Array objects to obtain their corresponding characters; for instance, we write $s[2]$ to obtain the third character of the string bound to s. However, String objects do not have explicit named properties corresponding to the indexes of the given string. In the example mentioned before, the String object bound to s does not actually have a property 2 storing its 3rd character. Hence, in order to model string indexed properties, String objects have their own `[[GetOwnProperty]]` internal method (making them exotic), which adds access to array-like zero-indexed name properties to String objects. In a nutshell, every time one accesses an index property of a String object, the ES engine will use the internal method `[[GetOwnProperty]]` of String objects to determine the character stored at that index, which is then returned in the form of a data property descriptor. The pseudo-code of the String object `[[GetOwnProperty]]` internal method is given in Figure 2.6.

15.5.5.2 [[GetOwnProperty]] (P) # Ⓣ Ⓔ

String objects use a variation of the [[GetOwnProperty]] internal method used for other native ECMAScript objects (8.12.1). This special internal method is used to add access for named properties corresponding to individual characters of String objects.

(...)

8. Let *resultStr* be a String of length 1, containing one character from *str*, specifically the character at position *index*, where the first (leftmost) character in *str* is considered to be at position 0, the next one at position 1, and so on.
9. Return a Property Descriptor { [[Value]]: *resultStr*, [[Enumerable]]: **true**, [[Writable]]: **false**, [[Configurable]]: **false** }

Figure 2.6: ES5 String's `[[GetOwnProperty]]` returning a new data property descriptor.

The importance of the String object immutability surfaces again, with the ability of indexing its characters. Listing 2.5 demonstrates a common pitfall for programmers who are unaware of the immutable characteristic of ES String objects. It should be noted that, executing the code of Listing 2.5 in strict mode [3] will throw a `TypeError` exception.

```
1 var s = "abc";
2 s[0] = "d";
3 s // "abc"
```

Listing 2.5: A common pitfall for programmers unaware of ES String object immutability.

Annex B of the ES5 standard [5] contains additional syntax and properties for some of the standard built-in objects, including the String object, for compatibility with past editions of the ES standard and/or some implementations of ES. This additional syntax and properties are not part of the ES5 standard, and thus this work does not cover them.

A mild curiosity regarding Unicode escape sequences, is that in Java, these escapes are processed before the compiler proceeds to lexical analysis [6]. This causes, for example, the Unicode escape sequence `\u000A`, which is the line feed, to be interpreted as an actual line terminator in the source text. If this escape sequence were to occur within a string literal, then the string would not have the closing quote. In ECMAScript, Unicode escape sequences are interpreted during lexical analysis, thus avoiding the unexpected behaviour described above (see Section 6 of the ECMA-262 Edition 5.1 [5]).

2.4 ES5 RegExp Object

The RegExp built-in object is part of the Text Processing category of the ES library. Regular expressions (RegExp) have their origin in automata theory and formal language theory. They were invented by the mathematician Stephen Cole Kleene during the 1950s and became popular due to their conciseness, simplifying pattern matching in text files and lexical analysis in compiler design. Every regular expression has an equivalent finite automaton, and every regular language can be defined by a regular expression. However, many modern RegExp libraries provide features that recognize non-regular languages, as is the case of the ES RegExp built-in object.

Figure 2.7 illustrates the internal representation of a typical ES5 RegExp object according to the ECMA-262 Edition 5.1 [5].

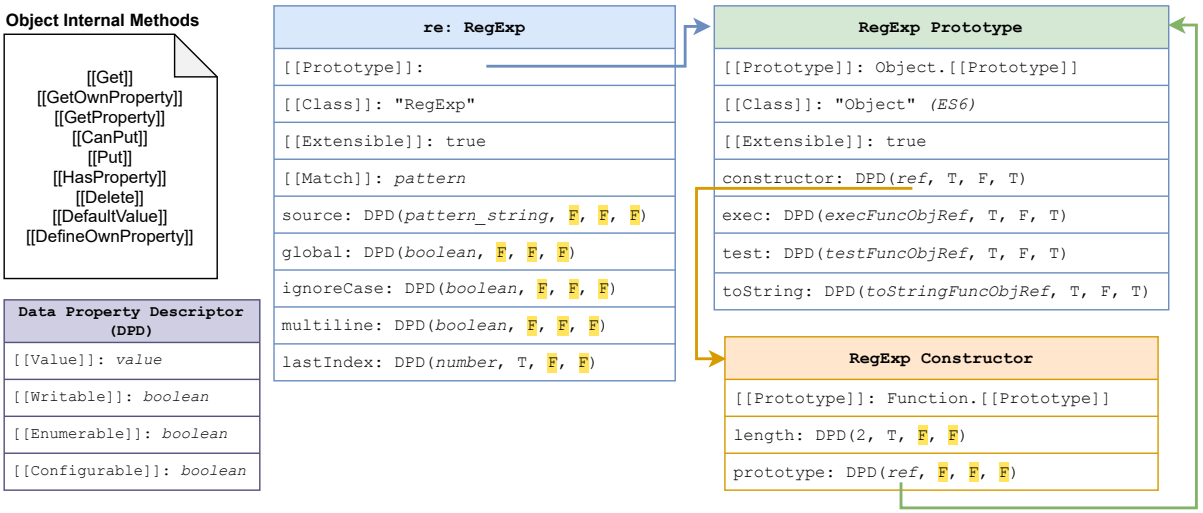


Figure 2.7: ES5 RegExp Object Graph.

RegExp instances have an internal property `[[Match]]` that holds an implementation dependent internal procedure that recognizes expansions of the corresponding regular expression. In the following

[6] Unicode Escapes in Java: `https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.3`

we will refer to this internal procedure as a *regular expression matcher*. The ECMA-262 standard specifies how regular expression patterns are to be transformed into regular expression matchers. RegExp instances also have the following named properties: `source`, holding a string representation of the pattern of the regular expression; `global`, a boolean value indicating if the search for the pattern on a text should look for all the matches, or stop at the first one found; `ignoreCase`, a boolean value indicating whether the search for the pattern on a text should be case-insensitive; `multiline`, a boolean value indicating whether or not the boundary characters `^ $` should match the beginning and ending of every line, instead of the beginning and ending of the whole text; and `lastIndex`, a number, coerced to an integer, that specifies the position in the text at which to start looking for a match.

Understanding patterns The syntax of ES regular expressions is quite complex. Here we only aim to give the reader a general idea of how they work. As an example, suppose that we are given an HTML element, which, for simplicity, may not contain HTML attributes nor other HTML elements within. We do not know the type of this HTML element nor its content, and we want to retrieve both. Listing 2.6 demonstrates a way to achieve this using the RegExp object.

```
1 const str = "<strong>I am recognized!</strong>";
2 const re = /<(.+)>(.*)<\/\1>/;
3 const [m, tag, content] = re.exec(str);
4 console.log(tag); // 'strong'
5 console.log(content); // 'I am recognized!'
```

Listing 2.6: Using the RegExp built-in object.

In this example, we make use of a RegExp literal (line 2) in order to define our pattern, so that we are exempt from escaping the occurring backslash (\) characters, as would be required in a string literal. RegExp literals must be enclosed in forward slash (/) characters. The evaluation of a RegExp literal yields a RegExp object. In our regular expression pattern, the following characters have special meaning: `( ) . + * \`. The backslash character is used, among other things, to escape special characters, making them literal. Occurrences of the forward slash character in a RegExp literal must be escaped in order to have literal meaning, instead of terminating the pattern. The dot (.) metacharacter matches any single character, except for the line terminator characters described in Table 3 of the ECMA-262 Edition 5.1 [5]. The plus sign (+) metacharacter matches the preceding element *one* or more times. In our pattern, we have a dot (.) metacharacter followed by a plus sign (+) metacharacter, which means that any character (except for line terminators) will be matched one or more times. This means that, if we ignore the parenthesis, for now, the pattern `<.+>` will match any HTML tag, considering that the characters `< >` have no special meaning, which is what we want. The asterisk (*) metacharacter matches the preceding element *zero* or more times, meaning that our pattern accepts an HTML element—with the previously mentioned restrictions—without any content. Notice that, because the dot (.) metacharacter does not contemplate the line terminator characters, the content of our HTML element must not contain any of these characters, if it is to be matched by our pattern.

By now, we have an idea of how pattern matching works, but we have yet to explain how we can extract specific information with a pattern, such as the HTML tag type and the content of the element. For this, we surround with parenthesis the sub-expression of the pattern whose match we want to store, forming a *capturing group*. When the `exec` method is called in line 3, it matches the string defined in line 1 with the RegExp pattern defined in line 2, returning an Array object containing the matched string in the first index and each matched capture group in the indexes that follow. For clarity, we use the destructuring assignment, in line 3, in order to assign each index of the returned array to a more descriptive variable.

Finally, we make use of a feature that exceeds regular languages, and that is the *backreference*, highlighted in blue. Backreferences allow us to match the same value that was matched by a specified capturing group, which requires the parser to store this value in order to recall it. Because our backreference has the number 1, it will match the same value that was matched by the first capturing group. This allows us to guarantee that the type of the HTML closing tag will equal the type of the opening tag.

RegExp grammar The grammar for regular expression patterns in ES5 is described in Section 15.10.1 of the ECMA-262 Edition 5.1 [5]. Due to its complexity and length, we have chosen to leave its description outside the scope of this document. However, it is worth noting that the grammar is not the same for all programming languages and libraries, and that implementations of regular expressions can vary between versions. Such is the case for ES, in which newer versions add new features to the RegExp built-in object.

2.5 ES5 JSON Object

JavaScript Object Notation (JSON) is a language-independent data interchange format based on a subset of JavaScript. It is primarily used for storing and transmitting data objects consisting of key-value pairs, arrays, and other serializable values [7], in a human-readable text format, although it can be used for other purposes, such as configuration files. The JSON format was originally specified by Douglas Crockford in the early 2000s and one of its earlier specifications was RFC 4627 [8], which was adopted by ES5 and is now obsolete. In 2013, JSON was standardized as ECMA-404 [13] and the specification used by ES6 was updated accordingly. This work focuses on the ES5 JSON object, but uses the JSON Data Interchange Syntax defined in ECMA-404.

Figure 2.8 illustrates the internal representation of the ES5 JSON object according to the standard.

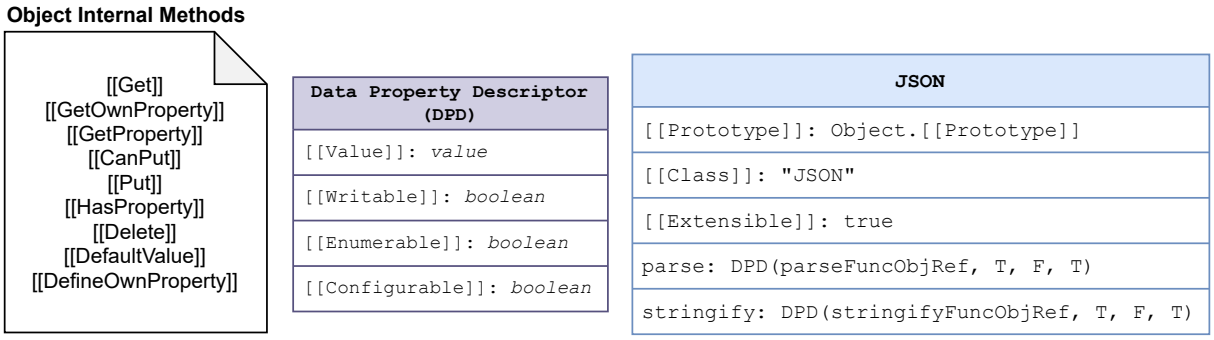


Figure 2.8: JSON Object Graph.

The JSON object is a single object that contains only two functions, `parse` and `stringify`, and has no constructor. Both functions are pure functions. The `parse` function parses a JSON text and produces an ES value, whereas the `stringify` function returns a string in JSON format representing an ES value.

To illustrate the advantages and shortcomings of the JSON format, Listing 2.7 shows a case where an Array object is serialized in a human-readable format, but in which data loss occurs.

[7] Serializable values in JSON include numbers, strings, booleans, arrays, objects (JSON objects), and null. Other ES data types, such as Date, Function, RegExp, and undefined, are not part of ECMA-404.

[8] RFC 4627: `https://www.ietf.org/rfc/rfc4627.txt`

```
1 Array.prototype[2] = 3;
2 var arr = [1, 2];
3 arr.prop = "I am not an ordinal named property!";
4 var jText = JSON.stringify(arr);
5 jText; // '[1,2]'
```

Listing 2.7: Data loss with the JSON `stringify` method.

The clear advantage of this format is that it can be understood by both human and machine. But the main disadvantage is that, as stated before, it cannot serialize all data types. As demonstrated by Listing 2.7, inherited properties of an object are lost, as well as non-ordinal named properties of Array objects. For this reason, JSON is not applicable for all purposes of serialization. When preserving the exact state of an ES object is required, other formats may be more suitable. For instance, ActionScript (AS), which is based on ECMAScript, uses the Action Message Format [9], a compact binary format, to serialize object graphs.

For the reverse process, that is, JSON deserialization, an unorthodox usage of the `parse` method is presented in Listing 2.8, whose behaviour we explain ahead with the help of ECMA-262's 5.1 Edition [5].

```
1  Array.prototype[1] = 3;
2
3  var json = "[1, 2]";
4  var arr = JSON.parse(json, function(key, value) {
5    if (key === "0") {
6      delete this[1];
7    }
8    return value;
9  });
10
11 arr; // [ 1, 3 ]
12 Array.prototype[1]; // 3
```

Listing 2.8: An unorthodox usage of the JSON `parse` method.

In line 1 of Listing 2.8, we define property `"1"` of the Array prototype object, setting its value to `3`, which all the Array objects inherit. In line 3, we define a JSON-formatted string, `json`, representing an Array object, whose property `"0"` contains `1` and property `"1"` contains `2`, with the latter shadowing the inherited property of the Array prototype object. In order to understand how the JSON's `parse` method works, the summary for this function is shown in Figure 2.9, as given in the ECMA-262's 5.1 Edition [5].

15.12.2 parse (text [, reviver]) # Ⓣ ® Ⓑ

The `parse` function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and **null**. JSON uses a more limited set of white space characters than *WhiteSpace* and allows Unicode code points U+2028 and U+2029 to directly appear in *JSONString* literals without using an escape sequence. The process of parsing is similar to 11.1.4 and 11.1.5 as constrained by the JSON grammar.

The optional *reviver* parameter is a function that takes two parameters, (*key* and *value*). It can filter and transform the results. It is called with each of the *key*/*value* pairs produced by the parse, and its return value is used instead of the original value. If it returns what it received, the structure is not modified. If it returns **undefined** then the property is deleted from the result.

Figure 2.9: An ECMA-262 note on JSON's *parse* method.

[9] Action Message Format, 31 October 2021 - `https://www.adobe.com/content/dam/acom/en/devnet/pdf/amf-file-format-spec.pdf`

Going back to Listing 2.8, we make use of the optional *reviver* parameter when we call the `JSON.parse` method (lines 4–9). The atypical usage of this method happens in our *reviver* function, when we delete property `"1"` of the newly constructed object once the *reviver* function is called for property `"0"` (lines 5–7). This leads us to question if we are deleting property `"1"` before or after it has been parsed, that is, if the *reviver* function is called for each property has soon as it is parsed, in which case the property `"1"` will not yet have been defined with the value of `2`, or if it is called for each property after the entire object has been parsed, in which case the property `"1"` will be deleted and the property of the Array prototype object will be passed to the *reviver* function instead.

The summary of Figure 2.9 is not clear on whether the *reviver* function operates during or after parsing. For clarification, we must consult the pseudo-code of the JSON `parse` method. Figure 2.10 shows the relevant portion of the algorithm to aid with our example. We can see that in line 2 of the pseudo-code, the JSON-formatted string is first parsed according to the grammar, and only afterwards the *reviver* function is called for each element of the Array through the recursive abstract operation `Walk`. Thus, when we attempt to delete the property `"1"` in line 6 of Listing 2.8, it has been already parsed and defined in the object. Consequentially, when the *reviver* function is called for this property, it will read the value that exists in the Array prototype object.

1. Let *JText* be ToString(*text*).
2. Parse *JText* using the grammars in 15.12.1. Throw a **SyntaxError** exception if *JText* did not conform to the JSON grammar for the goal symbol *JSONText*.
3. Let *unfiltered* be the result of parsing and evaluating *JText (...)*
4. If IsCallable(*reviver*) is **true**, then
   a. Let *root* be a new object *(...)*
   b. Call the [[DefineOwnProperty]] internal method of *root* with the empty String, the PropertyDescriptor {[[Value]]: *unfiltered*, *(...)*
   c. Return the result of calling the abstract operation Walk, passing *root* and the empty String. The abstract operation Walk is described below.

(...)

The abstract operation Walk is a recursive abstract operation that takes two parameters: a *holder* object and the String *name* of a property in that object. Walk uses the value of *reviver* that was originally passed to the above parse function.

1. Let *val* be the result of calling the [[Get]] internal method of *holder* with argument *name*.
2. If *val* is an object, then
   a. If the [[Class]] internal property of *val* is `"Array"`
      i. Set *I* to 0.
      ii. Let *len* be the result of calling the [[Get]] internal method of *val* with argument `"length"`.
      iii. Repeat while *I* < *len*,
         i. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and ToString(*I*).
         ii. If *newElement* is **undefined**, then
            1. Call the [[Delete]] internal method of *val* with ToString(*I*) and **false** as arguments.
         iii. Else
            1. Call the [[DefineOwnProperty]] internal method of *val* with arguments ToString(*I*), the Property Descriptor {[[Value]]: *newElement*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}, and **false**.
         iv. Add 1 to *I*.
   b. Else
      (Here Walk is called recursively for each property of a non-Array object.)
3. Return the result of calling the [[Call]] internal method of *reviver* passing *holder* as the **this** value and with an argument list consisting of *name* and *val*.

Figure 2.10: JSON's *parse* pseudo-code.

The *reviver* function is called for property `"1"`, even after it is deleted, because the abstract operation `Walk` iterates over all the properties of the parsed Array at parsing time, regardless of whether or not those properties are later deleted.

2.6 ES6 Promise Object

The Promise built-in object belongs to the Control Abstraction category of the ES library. Control Abstraction follows the DRY principle of software development, which means “Don’t Repeat Yourself”, wherein repeated software patterns are replaced with abstractions in order to make the code more readable and less verbose, so that the programmer may focus on the control flow and logic of the application, instead of low-level details. Common examples of control abstraction are higher order functions, closures, and lambdas.

In ES, Promise objects have greatly simplified the creation, combination, and chaining of asynchronous computations, eliminating the so-called “callback hell” of multiple nested callbacks[14], which can easily become difficult to follow. The Promise object owes its name to the fact that it is used as a place-holder for the eventual results of a deferred (and possibly asynchronous) computation—it is the promise of a result, that was either already settled in the past or still remains to be settled in the future. The Promise object works with a producer-consumer pattern, and it links the producing code to the consuming code. A Promise object is in one of three mutually exclusive states: “fulfilled”, “rejected”, and “pending”. If it is settled successfully we say that it is “fulfilled” (or resolved). If it is settled unsuccessfully we say that it is “rejected”. If a promise has not been yet settled, we say that it is “pending”.

Figure 2.11 illustrates the internal representation of a typical ES6 Promise object according to the ECMA-262 6th Edition [2], with a specific state depicting an example introduced further ahead.

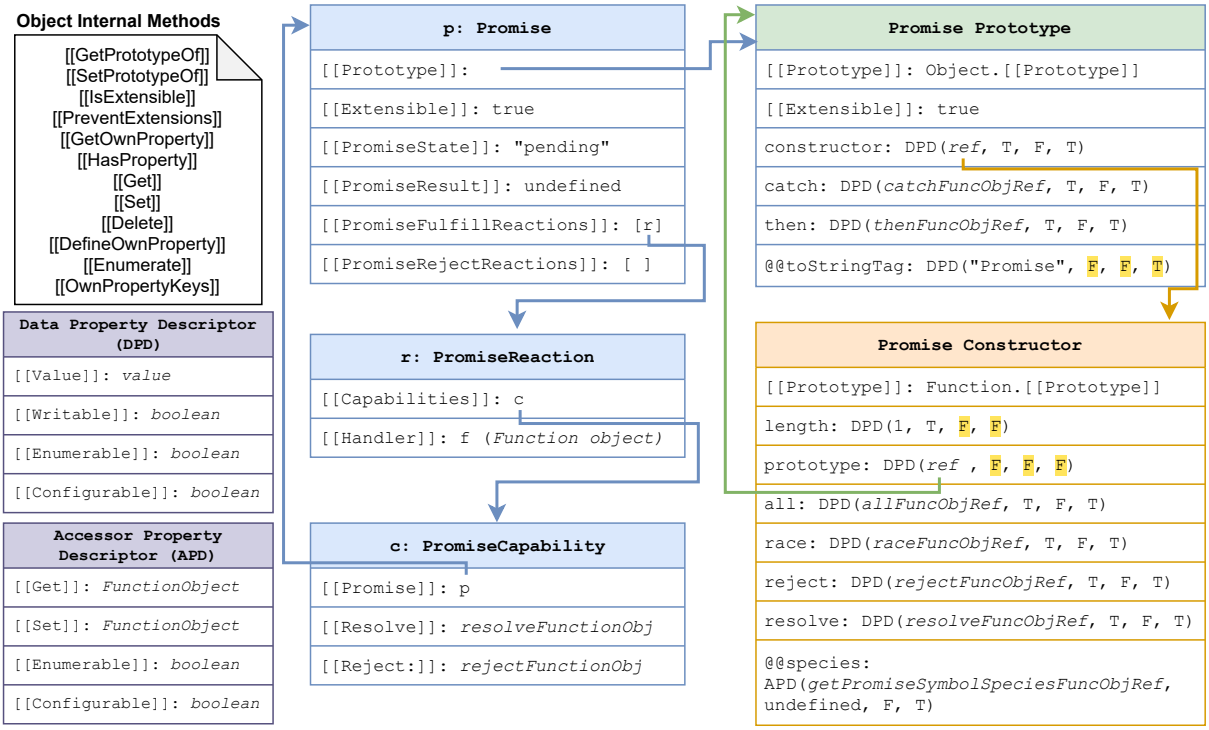


Figure 2.11: ES6 Promise Object Graph.

First, we would like to point out a few changes that occurred in the standard built-in ES objects from ES5 to ES6:

- The following internal methods that could be found in the standard built-in objects of ES5 have been removed: `[[GetProperty]]`, `[[CanPut]]`, `[[Put]]`, `[[GetProperty]]`, and `[[DefaultValue]]`.
- The following internal methods have been added to the standard built-in objects of ES6: `[[GetPrototypeOf]]`, `[[SetPrototypeOf]]`, `[[IsExtensible]]`, `[[PreventExtensions]]`, `[[Set]]`, `[[En-`

umerate]], and [[OwnPropertyKeys]].

- The [[Class]] internal property, which was common to all objects, has been removed.

ES6 introduces the Symbol type, which represents a non-String value that can be used as the key of an object property. In this work we do not cover the Symbol type, but we have included the Symbol keyed properties in Figure 2.11, which use the notation @@name, for completion.

Lastly, we introduce *accessor property descriptors* for the first time in this document. The difference between an accessor property descriptor and a data property descriptor, is that the latter accesses the value that it stores directly, whilst the former makes use of a [[Get]] function to retrieve the value and a [[Set]] function to update or set a new value. These functions may not be defined, however, in which case, attempting to use them will produce no result.

Understanding Promise objects by example In order to understand how Promise objects work, consider the example given in Listing 2.9, which exemplifies a deferred operation through the use of a Promise object.

```
1 function f(v) { console.log(v) }
2
3 var rf;
4 var p = new Promise((resolve, reject) => rf = resolve);
5 p.then(f);
6 rf(1);
7 f(2);
```

Listing 2.9: An example of a Promise object.

The Promise constructor receives as input an executor function, which captures the computation to be performed asynchronously. Executor functions have two arguments: a function resolve for stating that the corresponding promise has been resolved, and a function reject for stating that it has been rejected. Until one of these functions is called, the corresponding promise is left pending. In the example, the resolve function is called in line 6. Note that the created Promise is resolved outside the body of its executor function.

In terms of the producer-consumer pattern, the executor function is the producer, and the then method is the consumer. The first argument of the then method is a function that runs when the promise is resolved, receiving the value with which the Promise was resolved. The second argument of the then method is a function that runs when the promise is rejected, and receives the error. Both arguments to this method are optional. In our example, only the first argument is passed to the then method, the function f, which receives the result and prints it to the console.

Since we call the resolve function in line 6, one might expect the number '1' to be printed immediately, since the promise p has just been resolved. However, this is not the case. In order to understand why, we must first understand the ES execution model.

ES has a non-preemptive (or cooperative) concurrency model, meaning that each Job (2.6.1) runs to completion before the execution of a new Job. In this example, the top-level Job schedules the execution of function f once the Promise is resolved in line 6. However, the Job responsible for the execution of function f can only start executing once the top-level Job finishes. This explains why the number '1' will not be the first to be printed. The concepts of Job and Job Queue are formally introduced in the following definitions.

Definition 2.6.1 (Job)**.** *A Job is an internal function that initiates an ECMAScript computation when no other ECMAScript computation is currently in progress. Execution of a Job can be initiated only when there is no running execution context and the execution context stack is empty. A PendingJob is a*

request for the future execution of a Job. Once execution of a Job is initiated, the Job always executes to completion. No other Job may be initiated until the currently running Job completes. However, the currently running Job or external events may cause the enqueuing of additional PendingJobs that may be initiated some time after the completion of the currently running Job.

Definition 2.6.2 (Job Queue)**.** *A Job Queue is a FIFO queue of PendingJob records. Each Job Queue has a name and the full set of available Job Queues are defined by an ECMAScript implementation. Every ECMAScript implementation has at least the Job Queues* `ScriptJobs` *and* `PromiseJobs`*.*

We will now give a more detailed explanation of how the program given in Listing 2.9 is executed, detailing all the internal objects that it creates. The structure of Promise objects is complex. Figure 2.11 gives us the object graph associated with the promise `p` of the Listing 2.9 example after the execution of the `then` method (line 5), but before the promise gets settled (line 6). Each Promise stores its current state, reactions to be triggered when the promise is either resolved or rejected, and its result, in the internal properties `[[PromiseState]]`, `[[PromiseFulfillReactions]]`, `[[PromiseRejectReactions]]`, and `[[PromiseResult]]`, respectively. In the case of our example, the promise `p` is in the state "pending" and its result is undefined, as it has not been yet resolved. Observe that `f`, which is scheduled to execute after `p`, is not stored directly as a fulfill reaction. Instead, there is a PromiseReaction, `r`, which, in addition to storing `f` in its `[[Handler]]` property, also holds, in its `[[Capabilities]]` property, a PromiseCapability `c`, which stores the promise on whose settlement `f` should be executed (`c.[[Promise]]`), and the resolve and reject functions given to the executor function of that promise (`c.[[Resolve]]` and `c.[[Reject]]`). In the example, the promise capability `c` contains the promise `p` and the internal resolve and reject algorithms of the standard.

In Figure 2.12, we can observe the `FulfillPromise` and `TriggerPromiseReactions` abstract operations of the ECMA-262 6th Edition [2].

> **25.4.1.4 FulfillPromise (promise, value)**
>
> *(...)*
>
> 1. Assert: the value of *promise*'s [[PromiseState]] internal slot is **"pending"**.
> 2. Let *reactions* be the value of *promise*'s [[PromiseFulfillReactions]] internal slot.
> 3. Set the value of *promise*'s [[PromiseResult]] internal slot to *value*.
> 4. Set the value of *promise*'s [[PromiseFulfillReactions]] internal slot to **undefined**.
> 5. Set the value of *promise*'s [[PromiseRejectReactions]] internal slot to **undefined**.
> 6. Set the value of *promise*'s [[PromiseState]] internal slot to **"fulfilled"**.
> 7. Return TriggerPromiseReactions(*reactions, value*).
>
> **25.4.1.8 TriggerPromiseReactions (reactions, argument)**
>
> *(...)*
>
> 1. Repeat for each *reaction* in *reactions,* in original insertion order
>   a. Perform EnqueueJob(**"PromiseJobs"**, PromiseReactionJob, «*reaction, argument*»).
> 2. Return **undefined**.

Figure 2.12: `FulfillPromise` and `TriggerPromiseReactions` abstract operations.

The `FulfillPromise` abstract operation receives as parameters a Promise and a value. We can see that this function changes the Promise's state from "pending" to "fulfilled", sets both `[[PromiseFulfillReactions]]` and `[[PromiseRejectReactions]]` to undefined, sets `[[PromiseResult]]` to the `value` parameter, and finally calls `TriggerPromiseReactions`, which will schedule the fulfilled reactions of the given Promise to be executed.

The `TriggerPromiseReactions` abstract operation receives as parameters a list of Promise reactions and the value with which they were resolved. This function is responsible for scheduling all the reactions to be executed once the current Job ends. To this end, each reaction is enqueued into the `PromiseJobs` queue using the abstract operation `EnqueueJob`.

2.7 ES6 Incompatibilities with Prior Editions

Despite the attempt to keep ECMAScript backward compatible between versions, a few minor breaking changes were introduced in ES6. They are listed in Annex D and Annex E of the ECMA-262 6th Edition [2]. When testing our reference implementation against Test262 [3], we chose to apply the following minor changes from ES6 in order to pass a larger number of tests:

1. The named property `length` of Function objects is not configurable in ES5 (Section 13.2 of ES5, instruction 15), but it is configurable from ES6 onwards (Section 9.2.4 of ES6, instruction 3).
2. In ES5, the RegExp prototype object is itself a RegExp object (Section 15.10.6 of ES5), whereas in ES6 it is an ordinary object (Section 21.2.5 of ES6).
3. In ES5, the `exec` method of the RegExp prototype object returns null if the property `lastIndex` of the RegExp instance is set to a negative number (Section 15.10.6.2 of ES5, instruction 9.a.ii). However, in ES6, this property is set to 0 in case it is negative, and the function resumes with the attempt to find a match (Section 21.2.5.2.2 of ES6, instruction 4).
4. In ES5, the RegExp constructor throws a `TypeError` exception if it is given a RegExp object and a `flags` argument (Section 15.10.4.1 of ES5). In ES6, however, the `flags` argument is used instead of the flags from the given RegExp object in order to construct the new RegExp object (Section 21.2.3.1 of ES6, instruction 5.c).
5. ES5 specifies no order for retrieving the named properties of an object when iterating over them (Section 12.6.4 and Section 15.2.3.14 of ES5). However, ES6 specifies that integer index property keys be retrieved first, in ascending order, followed by the remaining String property keys, in property creation order, and, at last, followed by Symbol property keys, also in property creation order (Section 9.1.12 of ES6). This affects the behaviour of the `parse` and `stringify` methods of the JSON object.

Chapter 3

Related Work

The literature includes a great many number of works on different types of program analysis techniques for JavaScript, including: type systems [15, 16], points-to analyses [17], control-flow analyses [18], abstract interpretation [19, 20], information-flow analyses [21, 22, 23], and program logics [24, 11, 25]. Here we focus our account of the related work on projects that try to formalise the semantics of JS, including reference implementations of some of its built-in objects.

The particularity of our ES5/6 reference interpreter, when compared to previous projects, is that it is designed to be identical to the ECMA-262 specification. In contrast, existing reference interpreters/formalisations differ substantially from the text of the ECMA-262 standard, resulting in two important drawbacks:

- How can we know that we implemented the intended behaviour as specified in the ECMA-262 standard? Trust in reference implementations is only obtained through testing against Test262, which is known to have coverage issues.
- How to guarantee that the reference implementation is accessible to a wide audience comprising developers with very different programming backgrounds? Most existing reference implementations were developed in highly technical/mathematical formalisms (such as Coq [26] and K [27]), which are generally out of the reach of non-academic programmers.

The first formal operational semantics of JavaScript Maffeis et al. were the first to design an operational semantics for the ECMAScript language [28]. Their operational semantics targets the third version of the ECMA-262 standard and is written in small-step style [29]. This semantics was the first JS semantics to follow the ECMA-262 standard faithfully, modelling most of its internal functions and implicit behaviours. The authors use this semantics to reason about various security properties of web applications and mashups [30, 31]. However, the authors do not automate nor mechanise their semantics, leaving it as a long text document consisting of a large number of semantic rules written in the authors' own custom-made syntax.

Lambda calculi for reasoning about JavaScript code: λJS and S5 Guha et al. [32] define λJS, a core lambda calculus that captures the most fundamental features of the third version of the ECMA-262 standard. They implement an interpreter for λJS in Racket [33] together with a compiler from ES3 to λJS. However, the authors do not test their compiler against Test262 and only support a subset of ES3. This project also comes with a type system for checking a simple confinement property of λJS expressions. With this type system, the authors are able to verify simple confinement properties of JS programs by first compiling them to λJS, and then applying the type system to the resulting λJS programs.

Later, Politz et al. [34] develop S5, an extension of λJS from ES3 to ES5 with support for property descriptors, getters and setters, and a complete treatment of the `eval` statement. Analogously to λJS, S5 is implemented in Racket [33] and comes with a compiler from JS to S5 and an interpreter of S5 written in Racket. In contrast to λJS, which was not tested against Test262 [3], S5 was thoroughly tested against Test262, passing 8,157 tests out of a total of 11,606 ($\approx$70%). The authors report that most of the failing tests are targeted at built-in objects like RegExp and Date, which the authors only implemented partially (the authors state that they have only implemented 60% of the ES built-in objects). Furthermore, S5 also fails a significant number of tests aimed at non-strict code, showing that S5 is not entirely consistent with the ECMA-262 standard.

Mechanised semantics of JavaScript: JSCert and KJS Bodin et al. [35] implement JSCert, a formalisation of the semantics of ES5 written in the Coq interactive proof assistant [26]. The proposed semantics is written in pretty-big-step style [36]. Besides an operational semantics, the authors also implement a reference interpreter called JSRef, which they prove correct with respect to the defined operational semantics. Using the Coq-to-OCaml extraction mechanism [6], the authors were able to obtain an OCaml version of the JSRef interpreter, which they use to test JSRef against Test262 [3]. However, JSCert targets only strict code and does not support most of the ES5 built-in objects, only implementing a subset of the Global, Object, and Function built-in objects. JSRef passes 1,796 tests out of a total of 2,782 tests corresponding to the core features of the language. The authors argue that the failing tests make use of non-implemented features related to built-in objects. For instance, multiple tests targeting core features of the language make use of ES arrays, which are not supported by JSRef.

One year later, Gardner et al.[37] extend the JSRef interpreter with support for ES5 Arrays by linking it to the Google's V8 [38] Array built-in object implementation. Since this implementation is partially written in JS, the authors simply concatenate it to the given JS programs, directly implementing in OCaml the parts of the Array object that are not written in JS. The authors additionally provide a detailed account of the testing infrastructure used to evaluate the JSCert project, including a thorough breakdown of all passing and failing tests. One of the benefits of adding support for the Array built-in object is that it allows the authors to obtain better testing results with regard to the core features of the language that they do implement, as a great many number of tests for the core features of the language make use of arrays. More concretely, with the inclusion of the V8 Array built-in object, JSRef passes 2,440 core language tests out of 2,782, and 1,309 Array tests out of 2,267.

Park et al. present KJS [39], a formal semantics of ES5 written in the K framework [27], a state-of-the-art term-rewriting system with support for several types of program analyses. KJS was tested against Test262 passing 2,782 core language tests. The K framework includes a built-in symbolic analysis based on reachability logic [40]. Hence, by combining KJS with the symbolic facilities of the K framework, one can symbolically analyse JS programs. Later, the authors demonstrate how this strategy can be used in practice to reason about simple JS programs [41]. In particular, they use KJS and the K framework to verify various data structures and sorting algorithms implemented in JS, including: AVL tree, binary search tree, red-black tree, quick sort, bubble sort, insertion sort, and merge sort.

JSExplain Charguéraud et al. present JSExplain [42], a reference interpreter for the ES5 language that allows programmers to code-step not only their JS code but also the pseudo-code of the standard. JSExplain is implemented in a purely functional subset of OCaml [6], extended with a built-in monadic operator for automatically threading the state of the interpreter across pure computations [43]. The authors additionally implement a compiler from their purely functional fragment of OCaml to JS, allowing them to run JSExplain in the browser.

The main goal of JSExplain is to function as a JS code-stepper that allows the developer to code-step not only the code of their own program but also the pseudo-code of the ECMA-262 standard. To this end, JSExplain produces inspectable execution traces that bookkeep all the intermediate states generated by the execution of the JS interpreter. JSExplain was tested against Test262 [3], passing over 5,000 tests.

Operational semantics / modules of ES6 Promises Madsen et al. [44] design λP, a λ-calculus that captures the fundamental behaviour of ES6 Promises. More concretely, λP is an extension of λJS [32] with dedicated constructs for the creation and manipulation of Promises. The authors further introduce the concept of *Promise graph* and use it to reason about common bug patterns involving Promises. Importantly, the authors neither present a compiler from ES to λP nor an interpreter of λP, meaning that one cannot use the authors' formalism to execute real world ES programs that use Promises.

Later, Alimadadi et al. [45] develop PromiseKeeper, a runtime debugging tool built on top of Jalangi [46] for identifying and explaining Promise-related bugs. The authors apply PromiseKeeper to twelve Promise-based Node.js applications taken from GitHub, showing that PromiseKeeper is able to construct Promise graphs for real world applications with acceptable performance. Furthermore, PromiseKeeper was able to detect a variety of Promise related bugs, such as: missing reject reactions; attempt to settle a Promise multiple times; unsettled promises; and unnecessary promises.

Finally, Sampaio et al. [47] develop JaVerT.Click, an extension of JaVerT [11] with support for event-based programming. More concretely, JaVerT.Click includes ES reference implementations of: the ES6 Promise built-in object, the DOM Core Level 1 API [48], and the DOM UI Events API [49]. The authors use JaVerT.Click to symbolically test two real world event driven libraries: cash [50] and p-map [51]. JaVerT.Click is of special interest to us as its reference implementation of ES6 Promises follows, as we do, the ES6 standard line-by-line. However, JaVerT.Click implements ES6 Promises directly in JavaScript, while we do it in ECMA-SL. We believe that, as ECMA-SL is a much simpler language than JS, the integration of our reference implementation into the code base of JaVerT.Click would result in performance gains. To this end, one would simply need to compile our reference implementation to JSIL, the intermediate language of JaVerT [11].

JISET Very recently, the authors of [52] introduce JISET, an instruction set specifically designed to be a compilation target for ES code. They further develop an extraction mechanism that semi-automatically creates an ES to JISET compiler from the text of the ECMA-262 standard. This extraction mechanism generates not only compilation rules for all the constructs of the language, but also an ES parser. The JISET project targets the 10th edition of the ECMA-262 standard. The project comes with an execution engine for JISET which the authors use to test JISET against Test262, passing 18,064 out of 35,990 available tests.

Some aspects of the JISET project are quite similar to the ECMA-SL project, and to our project in particular. For instance, like JISET, we also developed a mechanism for automatically converting the text of the ECMA-262 standard into executable code; we target ECMA-SL, whilst JISET targets its own instruction set. Analogously to JISET, we also use custom-made regular expressions for code generation. An important difference between the two projects is that the JISET project does not support most of the ES built-in libraries, including the RegExp and JSON built-in objects, as well as the methods of the String object that interact with regular expressions. These are some of the most complex built-in objects whose implementation requires advanced programming language techniques, such as: parser construction and continuation-passing-style interpreters.

Contrasting with ECMA-SL Although there have been multiple implementations of the ECMA-262 standard, none of them feature a syntax that closely resembles the pseudo-code of the standard. Also, none of the reference implementations have implemented the ES JSON and RegExp built-in objects, as well as the ES String methods that depend on regular expressions. Additionally, most reference implementations ignore character encoding, unlike our reference implementation. Consequentially, `ECMARef5` is the most complete academic reference implementation of ES5 to date, passing 12,026 tests out of 12,074 filtered tests [53], whilst JSCert [35] passes 1,796 tests, KJS [39] passes 2,782 tests, JSExplain [42] passes 5,000 tests, S5 [34] passes 8,157 tests, and JS-2-JSIL [11] passes 8,797 tests. Even though JISET passes 18,064 tests, it only accounts for 62% of its available pool of tests (28,952), which is considerably larger than the pool of tests of `ECMARef5`, due to JISET targeting the 10th edition of the standard. Therefore, the existing reference interpreters prove themselves unsuitable for promoting an executable ES specification, which is the goal of ECMA-SL.

Chapter 4

Extending ECMA-SL

In this chapter, we start by giving an overview of ECMA-SL (4.1), where we describe the project's architecture and the language itself. We then describe our (partial) implementation of Unicode in ECMA-SL (4.2) and conclude with an overview of other minor extensions that we added to the language (4.3).

4.1 An Overview of ECMA-SL

The ECMAScript Specification Language (ECMA-SL) is a simple imperative language with top-level functions and extensible objects. It was created with the purpose of serving as a dedicated intermediate language (IL) for ES analysis and specification. For this reason, ECMA-SL contains all the control-flow constructs used by the pseudo-code of ECMA-262; these include, for instance, a `return` statement, a `do-while` loop, a `while` loop, and an `if` statement. As in ECMAScript, we can dynamically add / remove properties from objects. Finally, the primitive data types of ECMA-SL mostly coincide with those of ES, with the most notable difference being that ECMA-SL also includes the integer type.

ECMA-SL Project Architecture The ECMA-SL project contains four main components that together make up the ECMA-SL reference interpreter.

- `JS2ECMA-SL` - a tool written in *Node.js* that parses a given JavaScript program, using the *Esprima* parser [54], and then creates an ECMA-SL function, called `buildAST`, that builds the abstract syntax tree (AST) of the given program in memory, returning the ECMA-SL object corresponding to the root of the AST.
- `ECMARef5` - the ES5 interpreter written in ECMA-SL, which also contains the implementation of the ES5 built-in objects.
- `ECMARef6` - the ES6 interpreter written in ECMA-SL, which also contains the implementation of the ES6 built-in objects.
- ECMA-SL Execution Engine - the interpreter of ECMA-SL, written in the OCaml [6] programming language, which receives and executes an ECMA-SL program. The output of the program and execution trace are printed to the console; furthermore, the final heap resulting from the program's execution can also be serialised to a file.

Listing 4.1 shows the snippet of ECMA-SL code that feeds the AST generated by the function `buildAST` to the `ECMARef5` interpreter. Note that this program is the same for every given JS program. The only thing that changes is the code of the `buildAST` function, which is specifically generated for the

JS program to be executed. Hence, in order to execute a JS program in ECMA-SL, we simply need to generate its corresponding `buildAST` function, store it in the file `ES5_interpreter/ast.esl`, and run Listing 4.1 using our ECMA-SL execution engine. Figure 4.1 shows a diagram that represents the execution pipeline we have just described.

```
1 import "ES5_interpreter/ast.esl";
2 import "ES5_interpreter/ESL_Interpreter.esl";
3 function main() {
4     x := buildAST();
5     ret := JS_Interpreter_Program(x, null);
6     return ret
7 }
```

Listing 4.1: Interpreting a JS program's AST.

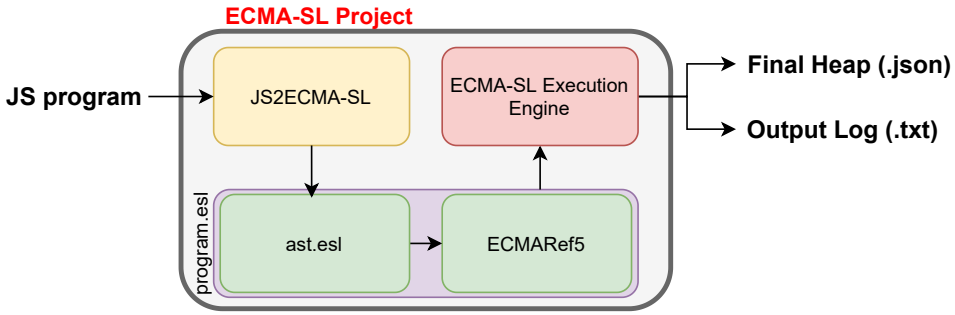


Figure 4.1: Architecture of the ECMA-SL project.

Besides the aforementioned tools, we also have a tool called `Heap2HTML` that creates an HTML representation of the obtained final heap. With this tool, we can visualise a Heap and navigate through the objects that it contains.

Extended ECMA-SL and Core ECMA-SL Beyond the scope of this thesis, although still relevant to understand the way ECMA-SL realizes its objectives, is the distinction between `Extended ECMA-SL`, simply referred to as ECMA-SL, and `Core ECMA-SL`. In order to be faithful to the pseudo-code of ECMA-262 for the purposes mentioned in Section 1.3, ECMA-SL contains several high-level constructs, such as *foreach* loops, that can be expressed using more fundamental constructs, such as *while* loops. For this reason, and with the aim of facilitating a static analysis of ECMA-SL code, a simpler version of ECMA-SL, called `Core ECMA-SL`, was also created. The ECMA-SL Execution Engine first compiles ECMA-SL code to `Core ECMA-SL` code, converting, for instance, *foreach* loops to *while* loops, and then interprets the obtained program. The compilation of ECMA-SL to `Core ECMA-SL` was implemented as part of another parallel thesis [53]. Figure 4.2 shows a diagram that represents the execution pipeline we have just described. Note that developing an interpreter for `Core ECMA-SL` is substantially simpler than doing so for the entire ECMA-SL language, and such has been done as part of a parallel MSc thesis [55] that focuses on the design of a dynamic information flow analysis for ECMA-SL. The complete grammars of ECMA-SL and `Core ECMA-SL` are given in Appendix A.

Understanding ECMA-SL by example In order for the reader to get acquainted with the ECMA-SL programming language, Listing 4.2 presents a simple ECMA-SL program that interacts with objects, lists, and tuples. Just as in ES, an object in ECMA-SL is a collection of properties, and a property is an association between a key and a value. Unlike in ES, however, ECMA-SL objects make no distinction

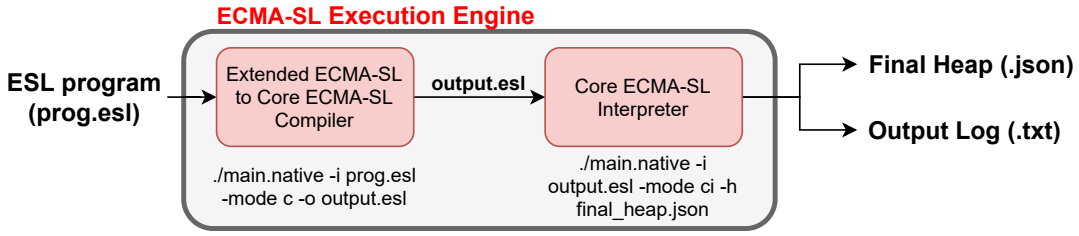


Figure 4.2: The ECMA-SL Execution Engine pipeline.

between named properties and internal properties, nor do they have property descriptors. ECMA-SL objects are implemented in OCaml using the OCaml Hashtbl [1] module. ECMA-SL lists are implemented in OCaml using the OCaml List [2] module, thus being an ordered sequence of elements, which in the case of ECMA-SL may have arbitrary data types. ECMA-SL tuples are also implemented in OCaml using the OCaml List [2] module, because, unlike ECMA-SL tuples, OCaml tuples are statically typed.

```
1  function Put(obj, key, value) {
2    obj[key] := value;
3    return
4  };
5
6  function Get(obj, key) {
7    return obj[key]
8  };
9
10 function GetKey(obj, value) {
11   /* When this function is called in line 32, L is:
12      [("key2", 2.), ("key1", 1), ("key3", "3"), ("Get1", 1)] */
13   L := obj_to_list obj;
14   i := 0;
15   len := l_len L;
16   while (i < len) {
17     tuple := l_nth(L, i);
18     K := fst tuple;
19     V := snd tuple;
20     if (value = V) {
21       return K
22     };
23     i := i + 1
24   };
25   return 'undefined
26 };
27
28 function main() {
29   obj := { key1: 1, key2: 2. };
30   Put(obj, "key3", "3");
31   Put(obj, "Get1", Get(obj, "key1"));
32   Put(obj, "GetKey2", GetKey(obj, 2.));
33   print obj.key4; /* 'undefined */
34   return
35 }
```

Listing 4.2: Working with ECMA-SL objects, lists, and tuples.

[1] OCaml Hashtbl, 31 October 2021 - `https://ocaml.org/api/Hashtbl.html`
[2] OCaml List, 31 October 2021 - `https://ocaml.org/api/List.html`

The ECMA-SL program given in Listing 4.2 corresponds to a simple implementation of a key-value dictionary. This program is composed of four functions: a function `main` that serves as the entry point of the program, a function `Put` for inserting a new key-value pair into the dictionary, a function `Get` for retrieving the value associated with a given key, and a function `GetKey` for retrieving the key associated with a given value. A dictionary is simply an ECMA-SL object, where keys are strings and the stored values may have arbitrary data types.

Starting with the `main` function, in line 29 we create an object using the same curly braces notation that is present in ES. The key names are coerced to strings. The most notable differences from ES are, however, the assignment operator (`:=`), the distinction between integers and floats [3], and the mandatory semicolon at the end of a statement, since semicolons are used as statement separators and not statement terminators.

In ECMA-SL, primitive data types are passed by value and objects are passed by reference. In line 30, we call the function `Put` which exemplifies this. This function receives an object, a key, and a value and inserts the key-value pair into the object, effectively mutating the object in-place. Note that although the logic of this function does not require the return of a value, it still requires a return statement in order to comply with the semantics of the language.

In lines 31 and 32, we call the functions `Get` and `GetKey`, respectively, and store the results as properties of the object, by using the `Put` function. The `Get` function is trivial, we retrieve the value paired to a key by using the key to index the object. For the `GetKey` function, we cannot use indexing and must loop through the object's properties in search for the value. ECMA-SL supports *for each* loops, but in this example we use a *while* loop in order to give more insight into the operations over ECMA-SL lists. First, we use the operator `obj_to_list` in order to convert our object to a list, so that we can loop through its properties. Then, we use the operators `l_len` and `l_nth` to get the length of the list and an element at an index of the list, respectively, effectively allowing us to loop over the list. Each element is represented by an ECMA-SL tuple, and we may access the first and second elements of the tuple, key and value, using the operators `fst` and `snd`, respectively. This way, we are able to compare the value for which we want to obtain the pairing key to each value that is present in the object, and return the first key which has a corresponding value that matches, or `'undefined` in case the value is not found.

As in ES, if a property of an object has not been defined, its value will be `'undefined` (line 33). Listing 4.3 shows the final heap of this program's execution.

```
1  {
2    "heap": {
3      "$loc_1": {
4        "GetKey2": "key2",
5        "key2": 2,
6        "key1": 1,
7        "key3": "3",
8        "Get1": 1
9      }
10   }
11 }
```

Listing 4.3: Final heap from executing Listing 4.3

[3] The Number type in ECMAScript uses a double-precision 64-bit format that complies with the IEEE Standard for Floating-Point Arithmetic (IEEE 754), for the most part – `http://es5.github.io/#x8.5`

4.2 Implementing UTF-8

An introduction to Unicode In order to replace language specific character encodings with one coordinated system, work began in the late 1980s on developing a "Universal Character Set" (UCS). The initial objective was to replace the typical 256-character encodings (e.g. extended ASCII), which required 1 byte per character, with one single encoding that would include all the required characters from most of the world's human languages, as well as symbols from technical domains. To this end, 2 bytes per character would be required—or so it was thought.

The encoding implementing the first version of Unicode was a 16-bit one, called UCS-2, which thrived from 1991 to 1995. As mentioned in Section 1.1, JavaScript was created in September 1995, having therefore adopted UCS-2 as the internal character encoding for strings. However, it became increasingly clear that 2^{16} (65,536) characters would not suffice [4], and UCS-4, which would require 4 bytes per character, was introduced. Evidently, UCS-4 was met with resistance, not only due to the waste of memory implicated, considering that a character that was previously encoded with 2 bytes now required 4 bytes, but also because popular technologies were already using a 2-byte-per-character encoding, as was the case of JS, Windows, and Java, to name a few. As a compromise, the 16-bit Unicode Transformation Format (UTF-16) encoding scheme, which is a variable-width encoding [5], was developed and introduced in July 1996 with version 2.0 of the Unicode standard.

Before we move on, it should be noted that, in Unicode, the term "character", which we have been roughly using so far to refer to the mapping of a code point (4.2.1), is not well defined [56]. Hence, we will hereinafter be using the terms code point and code unit (4.2.2) for the purposes of this work.

Definition 4.2.1 (Code point)**.** *Any value in the Unicode codespace; that is, the range of integers from* 0 *to* $10FFFF_{16}$*.* [6]

Definition 4.2.2 (Code unit)**.** *The minimal bit combination that can represent a unit of encoded text for processing or interchange. The Unicode Standard uses 8-bit code units in the UTF-8 encoding form, 16-bit code units in the UTF-16 encoding form, and 32-bit code units in the UTF-32 encoding form.* [7]

UTF-16 is backward compatible with UCS-2, and for this reason was adopted by JS and other software that was already invested in UCS-2. Code points less than 2^{16} are encoded with 2 bytes, which we call a 16-bit code unit, as in UCS-2. However, code points greater than or equal to 2^{16} are encoded with *surrogate pairs*—two 16-bit code units (4 bytes). There are 2048 16-bit code units that can be used to form a surrogate pair, which are code units that had not been yet assigned to characters in UCS-2, and there are $2\,048 * 2\,048 = 4\,194\,304$ surrogate pairs that could be formed with 2048 code units, but "only" $1\,024 * 1\,024 = 1\,048\,576$ pairs can actually be formed in UTF-16, for one simple reason: a parser should know if it is in the middle of a code point for efficient random access, meaning that UTF-16 is *self-synchronizing* on 16-bit code units. Hence, we have 1024 "high" surrogates ($D800_{16}$—$DBFF_{16}$) and 1024 "low" surrogates ($DC00_{16}$—$DFFF_{16}$). A surrogate pair consists of a high surrogate followed by a low surrogate, and the algorithm for determining the surrogate pair for a code point and vice versa is given in the UTF-16 FAQ [8]. Notice that, although more space-efficient than UCS-4, UTF-16 now has three disadvantages when compared to UCS-4:

1. Surrogate values cannot be mapped to "characters".
2. Unicode code points cannot be directly indexed.

[4] "What is UTF-16?", 31 October 2021 - `https://www.unicode.org/faq/utf_bom.html#utf16-1`
[5] A variable-width (or multibyte) encoding uses a varying number of bytes to encode different characters.
[6] Unicode's definition of code point, 31 October 2021 - `http://unicode.org/glossary/#code_point`
[7] Unicode's definition of code unit, 31 October 2021 - `http://unicode.org/glossary/#code_unit`
[8] "What's the algorithm to convert from UTF-16 to character codes?" `https://www.unicode.org/faq/utf_bom.html#utf16-4`

3. Cost of converting code points to and from a pair of surrogates.

The first disadvantage is insignificant in terms of code space size, because $2^{16} + 1\,024^2 - 2\,048 = 1\,112\,064$ code points are more than enough for the purposes of Unicode [9]. However, parsers must now be aware of invalid UTF-16 representations, or risk applications being susceptible to bugs and security vulnerabilities (e.g. CVE-2008-2938 [10], CVE-2012-2135 [11]). It should be noted that Unicode has reserved the UTF-16 surrogate range and prohibited code points greater than $10FFF_{16}$ in order to match with the constraints of UTF-16. Thus, UCS-4 with this limited subset has been defined as UTF-32.

Regarding the second disadvantage, it is not very common for a program to need to access the *nth* code point of a string without first inspecting the previous ones, and even in such a scenario, should there be combining characters (4.2.3) or grapheme [12] clusters (4.2.4) in the string, the perceived advantages of code point indexing in UCS-4/UTF-32 lose their meaning with respect to graphemes. And whilst truncation can be easier, it is not significantly so due to the self-synchronizing characteristic of UTF-16.

Definition 4.2.3 (Combining character)**.** *A character that modifies another character, such as diacritical marks. As an example, the grapheme* LATIN CAPITAL A WITH DIAERESIS*, Ä, can be represented with the code point* $00C4_{16}$ *or the sequence of code points* $0041_{16}0308_{16}$*, each mapping the* LATIN CAPITAL A *and the* COMBINING DIAERESIS*, respectively.*

Definition 4.2.4 (Grapheme cluster)**.** *A cluster of code points that form a grapheme. For instance, a Unicode emoji can consist of multiple code points [13].*

In order to address the third disadvantage, we must first realize that code points in the Unicode standard are divided into 17 planes, each plane being a contiguous group of 2^{16} (65,536) code points. The first plane, called the Basic Multilingual Plane (BMP), contains code points that map to characters for almost all modern human languages, and a large number of symbols. With the BMP being fully encoded by UCS-2, this means that, for the large majority of use cases, surrogate pairs will be rare occurrences in a string, and therefore the cost of conversion for practical purposes is negligible.

UTF-16, however, still has two major problems, with one of them affecting the ECMA-SL project considerably. The first problem being that UTF-16 is not backward compatible with ASCII, which a lot of software was using until the introduction of UCS-2, and much software kept using in spite of. So, for instance, consider a JS source file, encoded in ASCII, that is given to a JS interpreter. The string literals in this file will have to be converted to UTF-16 so that they can be manipulated by the interpreter internally. The second problem is that the byte order of both UTF-16 and UTF-32 depend on the endianness of the computer architecture, because I/O works in chunks of 2 bytes and 4 bytes respectively, which can result in ambiguity and may require a Byte Order Mark (BOM) for clarification.

With the goal of having a Unicode Transformation Format backward compatible with ASCII, UTF-8, also a variable-width encoding [5], was designed and first introduced in September 1992. Table 4.1 shows the structure of UTF-8, where the *x* characters are replaced by the bits of the code point. We can observe that the first 2^7 (128) code points are encoded exactly as in ASCII. And we can also observe a range of code points where UTF-8 is actually *less* space efficient than UTF-16 (0800_{16}—FFF_{16}). In addition, we notice that the number of leading 1's of a leading byte tells us how many bytes are used to encode a code point, with the exception of the ASCII range, and any byte starting with the sequence of bits 10 can only be a continuation byte, therefore making UTF-8 also *self-synchronizing*.

[9] "Will UTF-16 ever be extended to more than a million characters?": `https://www.unicode.org/faq/utf_bom.html#utf16-6`
[10] CVE-2008-2938: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2938`
[11] CVE-2012-2135: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2135`
[12] A grapheme is the smallest functional unit in written language.
[13] Full Emoji List, v13.1: `https://unicode.org/emoji/charts/full-emoji-list.html`

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 4.1: Structure of the UTF-8 encoding.

As a bonus, because UTF-8 works with 8-bit code units, it avoids the complications of endianness present in UTF-16 and UTF-32. Although, owing to the fact that Windows was one of the first operating systems to adopt Unicode, before UTF-8 was introduced, changing the internal representations of strings in its API from UCS-2 to UTF-8 would break a significant amount of existing code. Moreover, at that time, the need to extend the Unicode codespace with more than 2^{16} code points had not yet become apparent. Hence, code point indexing was considered to be an advantage of UCS-2, explaining why the designers of JavaScript also decided to opt for it.

As of 2021, UTF-8 is the *de facto* encoding for the World Wide Web, with over 97% of web pages using it [14]. Despite of this, ECMAScript never came to adopt UTF-8, deciding to keep backward compatibility with the first versions of ECMA-262.

String encoding in OCaml According to Chapter 6 of the ES5 standard [5], the source text of an ES program is to be interpreted using the UTF-16 encoding. If the source text happens to be saved in a different encoding, then it must first be converted to the UTF-16 encoding. This is due to the fact that string literals in ES are to be encoded in UTF-16. However, this detail had been overlooked in the beginning of the ECMA-SL project's development. In addition, the *ocamllex* [15] lexical analyzer, which had been chosen for the lexical analysis of ECMA-SL source code, has no support for Unicode.

Tackling the problem using UTF-8 In an attempt to address the problem of Unicode escape sequences occurring in string literals of some of the tests of Test262 [3], our first approach was to replace the existing string-related operators in ECMA-SL, which were unaware of the string encoding, with operators that assume the encoding to be UTF-8.

Former operator	New operator
`s_len`: Returns the number of bytes in a string.	`s_len_u`: Returns the number of UTF-8 encoded code points in a string.
`s_nth`: Returns the *nth* byte of a string.	`s_nth_u`: Returns the *nth* UTF-8 encoded code point of a string.
`s_substr`: Given a position `pos` and a length `len`, returns the substring of a string, starting at byte position `pos` and with a `len` length of bytes.	`s_substr_u`: Given a position `pos` and a length `len`, returns the substring of a string, starting at the UTF-8 encoded code point position `pos` and with a `len` length of UTF-8 encoded code points.
`from_char_code`: Returns a character from the given ASCII code.	`from_char_code_u` Returns a UTF-8 encoded code point from the given Unicode code.
`to_char_code`: Returns the ASCII code of a character.	`to_char_code_u`: Returns the Unicode code of a UTF-8 encoded code point.

Table 4.2: An overview of the ECMA-SL string-related operators.

[14] Usage survey of character encodings broken down by ranking, 31 October 2021, W3Techs.com - `https://w3techs.com/technologies/cross/character_encoding/ranking`

[15] Lexer and parser generators (ocamllex, ocamlyacc), 31 October 2021 - `https://ocaml.org/manual/lexyacc.html`

Table 4.2 displays the former operators and the operators that we implemented, with a short description for each. Besides implementing new operators, we have also added support for Unicode escape sequences in ECMA-SL, which are converted to UTF-8 encoded code points during lexical analysis. An observation that we have made regarding Unicode escape sequences, is that until ES6, a single Unicode escape sequence could only represent one code point in the BMP, more specifically, a single Unicode escape sequence would require exactly four hexadecimal digits (e.g. `\uhhhh`). This means that code points outside the BMP would have to be represented with surrogate pairs. From ES6 onwards, Unicode escape sequences support a new syntax that allows a variable number of hexadecimal digits, by enclosing them in curly braces (e.g. `\u{10FFFF}`, which is equivalent to `\uDBFF\uDFFF` in UTF-16).

In order to demonstrate how we deal with UTF-8 encoded code points, Listing 4.4 presents the source code of our OCaml function that receives a string and returns the number of UTF-8 encoded code points that it contains.

```
1  let s_len_u = fun (s : string) : int ->
2    let rec loop s cur_i_u cur_i =
3      if cur_i >= (String.length s) then (cur_i_u) else
4      let c = Char.code (s.[cur_i]) in
5        if (c <= 0x7f) then
6          loop s (cur_i_u + 1) (cur_i + 1)
7        else if c <= 0xdf then
8          loop s (cur_i_u + 1) (cur_i + 2)
9        else if c <= 0xef then
10         loop s (cur_i_u + 1) (cur_i + 3)
11       else
12         loop s (cur_i_u + 1) (cur_i + 4)
13   in loop s 0 0
```

Listing 4.4: Implementation of the ECMA-SL `s_len_u` operator in OCaml

In line 1 of Listing 4.4 we declare our function, `s_len_u`, which calls the recursive function `loop` in line 13. The recursive function `loop` (line 2) receives the current index of the UTF-8 encoded code point, `cur_i_u`, and the current index of the byte, `cur_i`, that we visit on each iteration. Recalling Table 4.1, the number of leading 1's of a leading byte tells us how many bytes are used to encode a code point. With this in mind, we increase `cur_i_u` and `cur_i` accordingly. For instance, if the current byte has a value greater than $7F_{16}$ (01111111_2) but less than or equal to DF_{16} (11011111_2), then our current byte, which is always the first of a UTF-8 encoded code point, must have two leading 1's, and thus we increase `cur_i` by two in order to get to the first byte of the next UTF-8 encoded code unit. Once there are no more bytes to visit in the string, `cur_i_u`, which now represents the number of UTF-8 encoded code points in the string, is returned. Notice that we do not check for invalid UTF-8 representations, because *Node.js*, which is used by `JS2ECMA-SL`, ensures that the ECMA-SL source text has a valid UTF-8 representation.

Besides the new operators presented in Table 4.2, we have also implemented operator `utf8_decode`, which receives a Unicode escape sequence and returns the corresponding UTF-8 encoded code point, and operator `hex_decode`, which receives an hexadecimal escape sequence (e.g. `\xhh`) and returns the corresponding UTF-8 encoded code point.

UTF-8 is a temporary solution The operators that we implemented deal with UTF-8 encoded code points. However, operations on ES strings should deal with 16-bit code units instead, which is not possible with the current UTF-8 encoded strings, due to the incompatibility between UTF-8 and UTF-16. For instance, the length of a UTF-16 encoded string should equal the number of 16-bit code units that it contains, which may or may not equal the number of UTF-8 encoded code points in a UTF-8 encoded string. For code points in the BMP, a 16-bit code unit of a UTF-16 encoded string is equal to the code

point, which means that for UTF-8 encoded strings containing only code points in the BMP, the number of UTF-8 encoded code points will be the same as the number of 16-bit code units if the string were to be encoded in UTF-16. But for code points in other planes, the operators that we implemented will not produce the desired results. Because all of the tests of Test262 [3] that target ES5 do not contain Unicode escape sequences outside the BMP, with the exception of one test for the JSON's `stringify` method, this solution works as a temporary work-around.

Proposed solution Due to time constraints, we were unable to implement an ideal solution in time for the Unicode problem. We considered making ECMA-SL's string-related operators aware of code points that would require two 16-bit code units in a UTF-16 encoding, but the fact is that iterating over code points is already detrimental to performance on its own. In ES and in most programming languages, string-related operations are only aware of code units, not code points. Thus, the length of a string is to be counted in code units and the *nth* "character" of a string is to be the *nth* code unit. Furthermore, the ECMA-SL source code of the `buildAST` function, that is constructed by `JS2ECMA-SL`, is encoded in UTF-8 in order to be compatible with *ocamllex* [15], and because `JS2ECMA-SL` is written in *Node.js*, isolated surrogate values are replaced with the $FFFD_{16}$ code point when written to a file, which is the replacement character recommended by the standard for invalid UTF representations [16].

In order to solve the Unicode problem, we propose the replacement of *ocamllex* [15] with *sedlex* [17], a lexer generator for OCaml [6], similar to *ocamllex*, that features support for UTF-16 inputs.

4.3 Other Extensions

Besides the string-related operators, we have also added other operators to ECMA-SL that were deemed necessary during the implementation of the ES5 built-in objects covered in this work.

- `int_to_string` - Converts an integer to a string. First used for creating and accessing index properties of the Array object, but eventually came to be used in several places.
- `int_of_string` - Converts a string representation of an integer to the integer.
- `floor` - Rounds a floating-point number to the integer value less than or equal to the number. First used to implement step 4 of the `reverse` method of the Array prototype, as defined in Section 15.4.4.8 of the ES5 standard [5].
- `l_prepend` - Adds an element to the beginning of a list. First used to implement step 4 of the `concat` method of the Array prototype, as defined in Section 15.4.4.4 of the ES5 standard [5].
- `l_reverse` - Reverses a given ECMA-SL list. Used to assist with sorting the properties of an object, for compatibility with the tests of Test262 [3] targeting the ES6 JSON built-in object.
- `l_remove_last` - Removes the last element of an ECMA-SL list. Used to implement step 11 of both the `JO` and `JA` abstract operations, defined in Section 15.12.3 of the ES5 standard [5].
- `octal_to_decimal` - Converts an octal value to a decimal value. Used to implement the `\xxx` RegExp meta-character, which is used to get the character specified by an octal number. Octal escape sequences occur in a few tests of Test262 [3], despite Section B.1 of Annex B of the ES5 standard [5] asserting that they have been removed from ES5.

[16] Are there any byte sequences that are not generated by a UTF? How should I interpret them?, 31 October 2021 - `https://www.unicode.org/faq/utf_bom.html#gen8`

[17] Sedlex, an OCaml lexer generator for Unicode, 31 October 2021 - `https://github.com/ocaml-community/sedlex`

- `int_to_four_hex` - Converts an integer to a string representation of four hexadecimal digits. Used to implement step 2.c.iii of the `Quote` abstract operation defined in Section 15.12.3 of the ES5 standard [5].
- `parse_number` - Given a string that starts with the representation of a JSON number as specified in Chapter 8 of the ECMA-404 [13] standard, extracts the part of the string that corresponds to the JSON number.
- `parse_string` - Given a string that starts with the representation of a JSON string as specified in Chapter 9 of the ECMA-404 [13] standard, extracts the part of the string that corresponds to the JSON string.

Chapter 5

Reference Implementation

This chapter elaborates on the reference implementation of the following ES built-in objects: ES5 Array (5.1), ES5 RegExp (5.3), ES5 String (partially, 5.2), ES5 JSON (5.4), and ES6 Promise (5.5). For each of these built-in objects, the internal representation, internal methods, implementation-dependent methods (if any), and auxiliary functions are presented.

5.1 ES5 Array

Internal representation In Figure 2.2, presented in Section 2.2, we had the opportunity to analyse the ES5 Array object's graph, whose corresponding implementation in ECMA-SL is represented by Figure 5.1. By comparing the two figures we can observe that they are similar. However, because ECMA-SL objects do not make a distinction between internal and named properties, the latter are stored in a property named `JSProperties`, while the former are stored as regular properties. This approach avoids name collisions between these two types of properties.

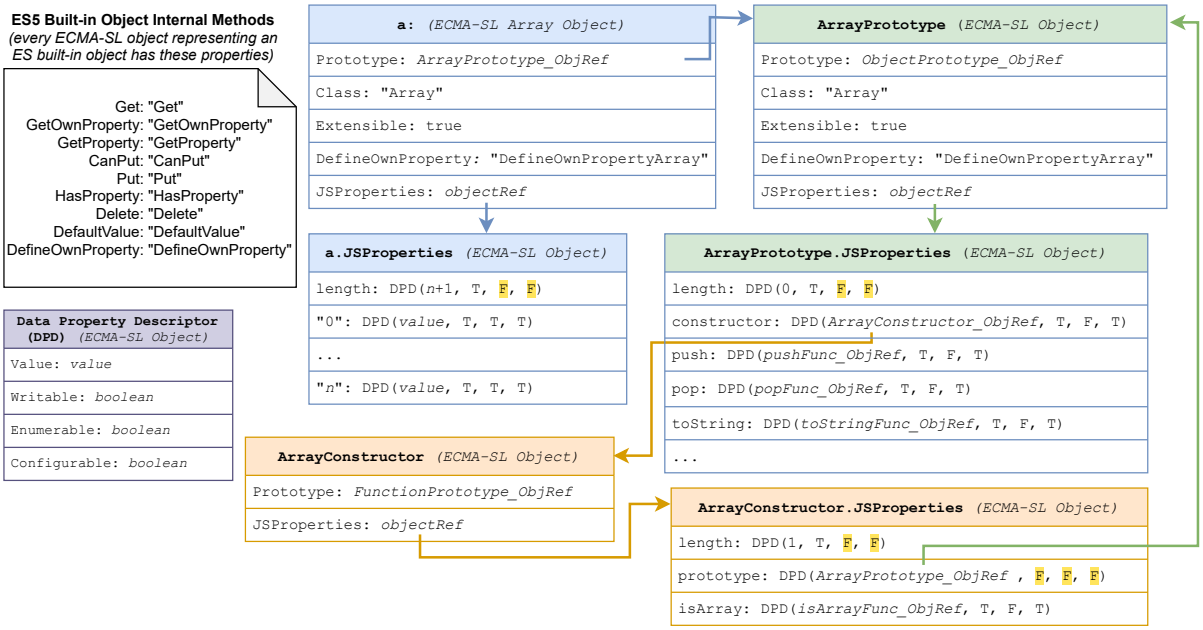


Figure 5.1: ES5 Array Object Graph in ECMA-SL.

Internal methods of an ES built-in object are stored as properties of its corresponding ECMA-SL object, with the key being the method's name, as described in ECMA-262, and the value being the

corresponding ECMA-SL function. Naturally, should there be variations of an internal method, the name given to its corresponding function can differ, as is the case of the `DefineOwnProperty` internal method of the ES Array object, whose corresponding ECMA-SL function we have decided to call `DefineOwnPropertyArray`. Built-in methods of an ES built-in object are treated differently. First, an ES Function object is created [1] that stores, among other data, the name of the ECMA-SL function for the corresponding built-in method. Then, a data property descriptor is created to store this Function object. Finally, the data property descriptor is assigned to a named property of the built-in object, whose key is, as expected, the name of the built-in method. Listing 5.1 shows part of the ECMA-SL code that demonstrates the process that we have just described.

```
1  function initArrayPrototype(global, objectPrototype, strict) {
2    prototype := NewECMAScriptObjectFull(objectPrototype, "Array", true);
3    setAllInternalMethodsOfObject(prototype);
4    prototype.DefineOwnProperty := "DefineOwnPropertyArray";
5
6    setJSProperty(prototype, "length", newDataPropertyDescriptorFull(0.,
       true, false, false));
7
8    /* 15.4.4.2 Array.prototype.toString ( ) */
9    toStringObject := CreateBuiltInFunctionObject([], "arrayToString",
       global, strict, null);
10   descriptor := newDataPropertyDescriptorFull(toStringObject, true,
       false, true);
11   setJSProperty(prototype, "toString", descriptor);
12
13   /* ... */
14
15   return prototype
16 };
```

Listing 5.1: Creating the Array Prototype object in ECMA-SL.

Line-by-line closeness In order to demonstrate the line-by-line closeness of our reference implementation of the ES5 Array object with respect to the text of ECMA-262, Listing 5.2 presents part of our ECMA-SL code for the variation of the `DefineOwnProperty` internal method, whose corresponding pseudo-code was introduced in Figure 2.4.

```
1  function DefineOwnPropertyArray(A, P, Desc, Throw) {
2    if (P = "length") {
3      /* ... */
4      while (newLen < oldLen) {
5        oldLen := oldLen - 1.;
6        deleteSucceeded := {A.Delete}(A, ToString(oldLen), false);
7        if (deleteSucceeded = false) {
8          newLenDesc.Value := oldLen + 1.;
9          if (newWritable = false) {
10           newLenDesc.Writable := false
11         };
12         DefineOwnProperty(A, "length", newLenDesc, false);
13         @Reject(Throw)
14       }
15     };
16   /* ... */
17 }
```

Listing 5.2: An ECMA-SL implementation of *DefineOwnProperty* for Array.

[1] Creating Function Objects in ES5: `http://es5.github.io/#x13.2`

Below, we explain the non-standard syntactic elements of ECMA-SL that occur in the snippet and that have not yet been introduced in this document.

- *Dynamic function calls:* The internal methods of an object, such as `Delete`, are stored as properties of the object. Hence, in order to call them, we first need to read the corresponding property. To this end, we make use of dynamic function calls, in which the name of the function to be called is only computed at runtime. In ECMA-SL, dynamic function calls are differentiated from static function calls through the use of curly braces; for instance, the expression `{A.Delete}(A, ToString(oldLen), false)` denotes a call to the function stored in the property *Delete* of the *Object* A.
- *Macros:* A macro resembles a function in syntax. However, a macro's invocation, preceded with the at sign, is expanded when compiled to `Core ECMA-SL`, effectively working by text substitution. Therefore, macros generally imply a faster runtime execution, as there is no overhead of a function call. And, more importantly, return statements in a macro become return statements of the invoking function. Listing 5.3 shows an example of the macro's definition for `Reject`.

```
1 macro Reject(Throw) {
2   if (Throw) {
3     throw TypeErrorConstructorInternal()
4   } else {
5     return false
6   }
7 };
```

Listing 5.3: `Reject` macro in ECMA-SL.

Implemented methods Table 5.1 shows the methods related to the ES5 Array object that were implemented in ECMA-SL as part of this work. For each method a brief description is given along with the lines of code (LOC) required to implement it. The ES5 Array object was fully implemented.

Section	Name	Description	LOC
		Set up Array's Object Graph	
15.4.3	`initArrayObject`	Creates the Array constructor object and calls `initArrayPrototype` to create the Array prototype object. Adds the `isArray` function and the Array prototype object as properties of the Array constructor and sets the constructor as a property of the Array prototype.	12
15.4.4	`initArrayPrototype`	Creates the Array prototype object, assigning all the methods under section 15.4.4 to itself, among other properties, as demonstrated in Listing 5.1.	69
		Auxiliary methods	
15.4	`isSparseArray`	Checks if an Array object is sparse. An Array object is said to be sparse if any value of its indexes between 0 (inclusive) and *length* (exclusive) is undefined.	12
15.4	`isArrayIndex`	Checks if a property of the Array object is an index, as described in the standard.	5

	`getArrayPrototype`	Gets the Array prototype object from the Array constructor object assigned to the Global object.	7
	`setAllInternalMethodsOfArray`	Assigns the default internal methods to the Array object, as well as the variation of the `DefineOwnProperty` method.	5
		Array constructor	
15.4.2	`ArrayConstructor`	The Array constructor function, selects either `internalNewArray` or `internalNewArrayLen`, based on the number of arguments.	8
15.4.2.1	`internalNewArray`	Constructs an Array object if no arguments or at least two arguments were passed to `ArrayConstructor`. Each argument becomes an element of the array, by order passed.	16
15.4.2.2	`internalNewArrayLen`	Constructs an Array object if exactly one argument was passed to `ArrayConstructor`. If the argument is a number, the Array's length is set to it. Otherwise, the argument becomes an element of the Array.	19
		Methods of the Array constructor	
15.4.3.2	`isArray`	Checks if the argument is an Array object.	10
		Methods of the Array prototype	
15.4.4.2	`toString`	Calls the `join` method of the Array object if it is defined. Calls `Object.prototype.toString` (15.2.4.2) otherwise.	8
15.4.4.3	`toLocaleString`	Converts the elements of the Array object to strings, using their `toLocaleString` methods, and concatenates these strings using a *separator* string derived in an implementation-defined locale-specific way, which in our case is the comma character regardless of locale.	40
15.4.4.4	`concat`	Returns an Array resulting of the concatenation of the current Array object with the objects passed as arguments. Each element of an Array passed as argument will be added to the resulting Array object.	33
15.4.4.5	`join`	Converts the elements of the Array to strings and concatenates them, separating them with the *separator* argument, which, if not provided, shall be a comma.	32
15.4.4.6	`pop`	Removes and returns the last element of the Array object.	18
15.4.4.7	`push`	Appends the arguments to the end of the Array object and returns the new length of the Array object.	14
15.4.4.8	`reverse`	Reverses the order of the elements in the Array object.	31
15.4.4.9	`shift`	Removes and returns the first element of the Array object.	27
15.4.4.10	`slice`	Receives indexes `start` and `end` as arguments and returns a new Array containing the elements of the original Array from `start` (inclusive) to `end` (exclusive).	37

15.4.4.11	`sort`	By default, sorts the Array in ascending order and compares the elements as strings. A callback can be passed as argument and it should receive two arguments to compare. The sorting algorithm is implementation-dependent—we chose Quicksort.	72
15.4.4.12	`splice`	This method can be used to delete and/or add elements at any position of the Array object. Receives a `start` index argument, a `deleteCount` argument, and an optional list of items. Deletes `deleteCount` elements starting at index `start`. The optional items are added after the `start` index.	74
15.4.4.13	`unshift`	Prepends the arguments to the start of the Array object by order.	30
15.4.4.14	`indexOf`	Searches for an element in the Array object, in ascending order, using the Strict Equality Comparison Algorithm (11.9.6). Returns the index of the first occurrence or -1 if it is not found.	42
15.4.4.15	`lastIndexOf`	Similar to `indexOf` (15.4.4.14), but searches in descending order.	37
15.4.4.16	`every`	Executes a callback for each element in the Array that is not undefined. The callback returns a value that is coercible to a boolean. If the callback returns false once then `every` returns false, otherwise `every` returns true.	29
15.4.4.17	`some`	Similar to `every` (15.4.4.16), but returns true if the callback returns true for at least one element.	29
15.4.4.18	`forEach`	Executes a callback for each element in the Array that is not undefined, in ascending order.	26
15.4.4.19	`map`	Executes a callback for each element in the Array that is not undefined and returns a new Array with the results.	29
15.4.4.20	`filter`	Executes a callback for each element in the Array that is not undefined and returns a new Array with the elements for which the callback returned *true*.	33
15.4.4.21	`reduce`	Executes a callback, which receives the result from the previous call to the callback, for each element in the Array that is not undefined, in ascending order. The accumulated value is returned.	41
15.4.4.22	`reduceRight`	Similar to `reduce` (15.4.4.21), but executes the callback in descending order.	41
		Internal methods of Array instances	
15.4.5.1	`DefineOwnPropertyArray`	A variation of the method given for other native ES objects (8.12.9). In particular, it describes the steps for when the *length* property or an index property of an Array object is updated.	65
Total:			951

Table 5.1: Implemented methods of the ES5 Array object.

Implementation-defined behaviour Some methods, or part of their behaviour, can be implementation-defined. This is the case of the `toLocaleString` and `sort` methods of the Array prototype object. The `toLocaleString` method, for instance, has the following step described in ECMA-262 Edition 5.1 [5]:

> Let *separator* be the String value for the list-separator String appropriate for the host environment's current locale (this is derived in an implementation-defined way).

For simplicity, we always set *separator* to the comma character. However, it is interesting to note that the list-separator String value can, in theory, be any string.

The `sort` method is more complex. There are many conditions of this method that are implementation-dependent, such as:

- Elements deemed as equal by the comparison function do not necessarily remain in their original order. This is the case with our implementation.
- If an index property is a data property whose `[[Writable]]` or `[[Configurable]]` attribute is false, or if it is an accessor property. In our implementation we do not handle these conditions in any specific way.
- The sorting algorithm is implementation dependent. We chose Quicksort for its simplicity.

Implementation-defined behaviour is outside the scope of the goals of ECMA-SL, thus, we approach these cases in a minimalistic way.

5.2 ES5 String

Internal representation Figure 2.5 gave us an overview of the ES5 String object's graph. We now have the opportunity to analyse the corresponding implementation in ECMA-SL, illustrated in Figure5.2.

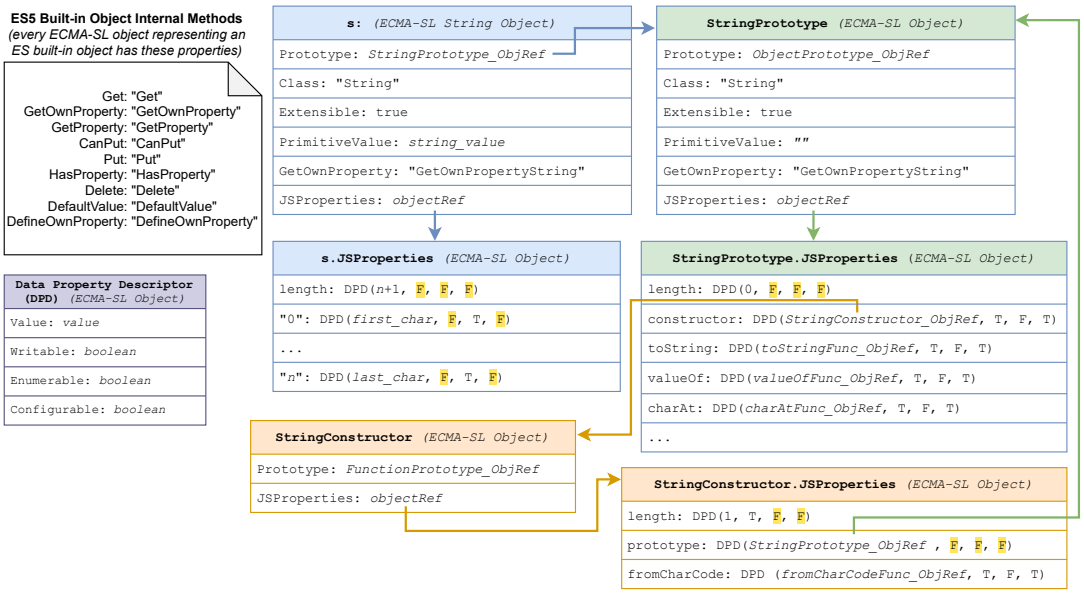


Figure 5.2: ES5 String Object Graph in ECMA-SL.

The difference that stands out in this implementation, aside from the differences covered in Section 5.1, is that we actually store each character of the string in a named property of the object, despite

the fact that the `[[GetOwnProperty]]` internal method creates and returns a new data property descriptor for a given index. We take this approach because the auxiliary functions in ECMA-SL that had been created to get the named properties of an object will look for them in the object's `JSProperties` property. Therefore, statements such as the *for-in* loop, that iterate over the enumerable properties of an object, can only be aware of them if they exist in the `JSProperties` property of the object being iterated over.

Line-by-line closeness Listing 5.4 shows the implementation of the pseudo-code instructions of the `[[GetOwnProperty]]` internal method previously presented in Figure 2.6. In this method, we make use of the `s_nth_u` operator, which is described in more detail in Section 4.2. However, this operator retrieves the *nth* code point of the string, when it should retrieve the *nth* 16-bit code unit.

```
1 function GetOwnPropertyString (S, P) {
2   /* ... */
3   /* 8. Let resultStr be a String of length 1, containing one character
      from str, specifically the character at position index, where the
     first (leftmost) character in str is considered to be at position 0,
      the next one at position 1, and so on. */
4   resultStr := s_nth_u (str, int_of_float index);
5   /* 9. Return a Property Descriptor { [[Value]]: resultStr, [[
     Enumerable]]: true, [[Writable]]: false, [[Configurable]]: false }
     */
6   return newDataPropertyDescriptorFull(resultStr, false, true, false)
7 };
```

Listing 5.4: String's `[[GetOwnProperty]]` internal method in ECMA-SL.

The patterns of some pseudo-code instructions have very few occurrences in the standard. For instance, the pattern of the pseudo-code's instruction eight occurs only in one other function of the ES5 standard [5]—the `charAt` method of the String built-in object.

Implemented methods All methods of the ES5 String object were implemented in ECMA-SL, but Table 5.2 only shows the methods that were implemented as part of this thesis. For each method, a brief description is given along with the lines of code (LOC) required to implement it.

Section	Name	Description	LOC
		Methods of the String prototype	
15.5.4.10	`match`	Matches the string against a regular expression and returns the matches as an Array object. If the *g* (global) modifier is not set in the RegExp object, then only the first match is returned.	47
15.5.4.11	`replace`	Receives a `searchValue` parameter, which can either be a string or a regular expression, and a `replaceValue` parameter, which can either be a string or a function. Occurrences of `searchValue` in the string are replaced with `replaceValue`, or the value returned from calling it. When `replaceValue` is a string, it can have replacement text symbol substitutions as described in Table 22 of the ES5 standard [5].	94
15.5.4.12	`search`	Matches the string against a regular expression and returns the position of the first match.	17

15.5.4.14	`split`	Splits a string using the `separator` parameter, which can be a string or a regular expression, and returns an Array object with the sub-strings. The optional `limit` parameter sets the maximum amount of sub-strings that the resulting Array may contain.	77
15.5.4.16	`toLowerCase`	Returns the string converted to lower-case. The Unicode character mappings are found in the `UnicodeData.txt` and `SpecialCasings.txt` files of the Unicode character database [57].	49
15.5.4.17	`toLocaleLowerCase`	The same as `toLowerCase`, except that some characters are supposed to be converted according to the host environment's locale, which may conflict with the Unicode case mappings in some cases. Due to time constraints and lack of coverage from Test262 [3], our implementation of this method is identical to the `toLowerCase` method.	
15.5.4.18	`toUpperCase`	Returns the string converted to upper-case. The implementation is similar to the `toLowerCase` method's, although we left out some conditional casings, due to time constraints and lack of coverage from Test262 [3].	29
15.5.4.19	`toLocaleUpperCase`	Identical to `toUpperCase`. Just as in `toLocaleLowerCase`, we do not consider the host environment's locale.	
15.5.4.20	`trim`	Returns the string with white-space characters removed on both ends. The white-space characters are specified in Table 2 and Table 3 of the ES5 standard [5].	36
		Abstract methods	
15.5.4.14	`SplitMatch`	Used by the `split` method in order to get a match result from the string.	20
		Auxiliary methods	
	`resolveDollars`	Used by `replace` in order to replace text symbols as specified in Table 22 of the ES5 standard [5].	87
	`isSpaceCharacter`	Used by `trim` in order to verify if a character is one of the white-space characters described in Table 2 or Table 3 of the ES5 standard [5].	12
Total:			468

Table 5.2: Implemented methods of the ES5 String object.

Aside from the methods that were fully implemented in the context of this thesis, we updated other methods of the ES5 String built-in object in order to make use of the new ECMA-SL operators described in Section 4.2. Although these operators deal with Unicode code points instead of 16-bit code units, they essentially produce the same result for strings containing only code points in the BMP, thus allowing us to pass more tests of Test262 [3], as a temporary solution for the lack of UTF-16 support in ECMA-SL.

Implementation dependent methods The String built-in object has several functions whose summary, or pseudo-code, does not rigorously specify how to implement. Such is the case for the following casing operations: `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, and `toLocaleUpperCase`. For these methods, the specification tells us to use the case mappings in the Unicode character database [57], that is, the `UnicodeData.txt` and `SpecialCasings.txt` files included in this database. However, this mapping implementation will produce several lines of code due to the conditional mappings that can be found in the `SpecialCasings.txt` file. As an example, when converting the upper-case Σ character ($03A3_{16}$) to lower-case, it will generally be mapped to σ ($03C3_{16}$), unless the character appears at the end of a string, in which case it will be mapped to ς ($03C2_{16}$). This was the only conditional casing that was implemented in this work, required for passing all tests of Test262 [3] that cover the ES5 standard [5].

An observation that we have made while implementing the aforementioned casing methods, is that there is an additional mapping possible, besides the lower-case and upper-case mappings, and that is the title-case mapping. For instance, the lower-case ß character ($00DF_{16}$) is mapped to the upper-case `SS` ($0053_{16}0053_{16}$) and to the title-case `Ss` ($0053_{16}0073_{16}$). This leads us to suggest the addition of a `toTitleCase` method to the String prototype object.

5.3 ES5 RegExp

Internal representation In Figure 2.7, presented in Section 2.4, we had the opportunity to analyse the ES5 RegExp object's graph, whose corresponding implementation in ECMA-SL is represented by Figure 5.3.

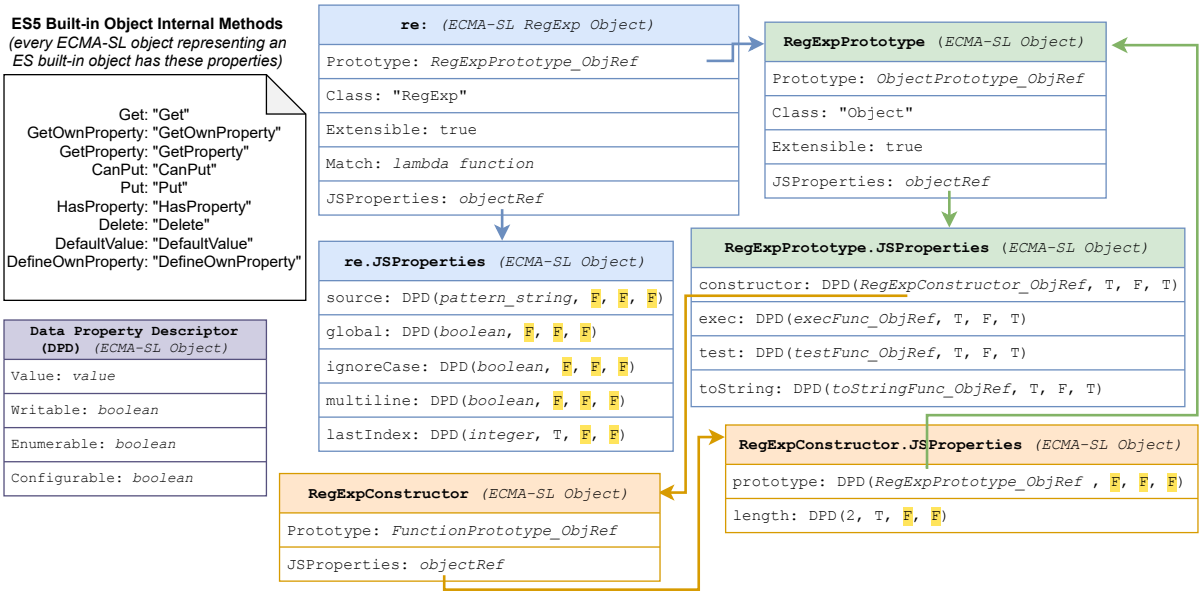


Figure 5.3: ES5 RegExp Object Graph in ECMA-SL.

In ECMA-SL, ES5 RegExp instances are represented as ECMA-SL objects. The methods shared by all RegExp instances are stored in the ECMA-SL object corresponding to the `RegExp.[[Prototype]]` object. The named properties of a RegExp instance, `re`, are stored in the object `re.JSProperties`. Particularly, RegExp instances store their corresponding matchers in the internal property `[[Match]]`. In ECMA-SL, matchers are modelled as lambda functions that recognize expansions of the corresponding regular expression. Our implementation includes a regular expression interpreter that, given the AST

of a regular expression, generates the lambda function corresponding to its matcher. The implemented interpreter follows the algorithms proposed by Fragoso Santos et al. in [58]. An account of these algorithms is out of the scope of this thesis. The AST of the regular expression to be interpreted is computed by JS2ECMA-SL using the regexp-tree [2] regular expression parser.

Line-by-line closeness Listing 5.5 shows a portion of the ECMA-SL code that corresponds to the loop performed by the exec method for finding a match in a given string. Each instruction is accompanied by a comment with its corresponding pseudo-code instruction, as described in the ES5 standard. As before, we can see that the ECMA-SL constructs are descriptive and very similar to the pseudo-code of the standard. Furthermore, the call to the [[Match]] internal method is abstracted in our code, therefore not conflicting with the goals of ECMA-SL, as far as the exec method is concerned.

In this loop, the algorithm starts with the matchSucceeded boolean value set to false. If the i integer value, corresponding to the value stored in the property lastIndex of the RegExp instance, is negative or larger than the length of the string, then the property lastIndex is set to zero and the exec method returns null. However, it should be noted that, as explained in item 3 of the enumeration given in Section 2.7, we chose to modify this behaviour in order to comply with ES6, which we have done by setting lastIndex to zero, should it be negative, before the execution of this loop. If the i value is within the range of indexable characters of the string argument, then the matcher for this regular expression is called with the string and i—the index of the string where the regular expression will attempt to match. If the match fails, then the index i is increased and the next iteration of the loop starts, until either a match is found or lastIndex is set to zero and null is returned.

```
1  /* 9. Repeat, while matchSucceeded is false */
2  while (matchSucceeded = false) {
3    /* a. If i < 0 or i > length, then */
4    if ((i < 0) || (i > length)) {
5      /* i. Call the [[Put]] internal method of R with
6          arguments "lastIndex", 0, and true. */
7      {R.Put}(R, "lastIndex", 0., true);
8      /* ii. Return null. */
9      return 'null
10   };
11   /* b. Call the [[Match]] internal method of R with
12       arguments S and i. */
13   ret := {R.Match}(R, S, i);
14   /* c. If [[Match]] returned failure, then */
15   if (isFailure(ret)) {
16     /* i. Let i = i+1. */
17     i := i + 1
18   }
19   /* d. else */
20   else {
21     /* i. Let r be the State result of the call to [[Match]]. */
22     r := ret;
23     /* ii. Set matchSucceeded to true. */
24     matchSucceeded := true
25   }
26 };
```

Listing 5.5: A snippet of the ECMA-SL implementation of the RegExp prototype's exec method.

[2] regexp-tree, 31 October 2021 - https://github.com/DmitrySoshnikov/regexp-tree

Implemented methods Table 5.3 shows the methods related to the ES5 RegExp object that were implemented in ECMA-SL as part of this work, with the exception of the auxiliary methods, due to their large number and dissociation from the goals of ECMA-SL. For each other method, a brief description is given along with the lines of code (LOC) required to implement it. The ES5 RegExp object was fully implemented.

Section	Name	Description	LOC
		Set up RegExp's Object Graph	
15.10	`initRegExpObject`	Calls `initRegExpConstructor` to create the RegExp constructor object and calls `initRegExpPrototype` to create the RegExp prototype object. Returns the RegExp constructor object.	7
15.10.5	`initRegExpConstructor`	Creates the RegExp constructor Function object and sets its `prototype` property to the RegExp prototype object.	6
15.10.6	`initRegExpPrototype`	Creates the RegExp prototype object, setting all of its internal and named properties.	32
		RegExp constructor	
15.10.3	`RegExpConstructorCalledAsFunction`	Receives a `pattern` parameter and a `flags` parameter. If `pattern` is a RegExp object and `flags` is undefined, then the RegExp object is returned. Otherwise it calls `newRegExp`.	7
15.10.4	`RegExpConstructor`	Calls `RegExpConstructorCalledAsFunction` if the `this` object is null or undefined. Otherwise it calls `newRegExp`.	11
15.10.4.1	`newRegExp`	Constructs a RegExp object, given the arguments `pattern` and `flags`. Calls the auxiliary function `parsePattern`.	28
		Methods of the RegExp prototype	
15.10.6.2	`exec`	Receives a string argument, matches the string against the regular expression, and returns an Array object that contains the results of the match, or null if the string did not match.	60
15.10.6.3	`test`	Performs the `exec` method on a given string, returning true if the result is not null and false otherwise.	9
15.10.6.4	`toString`	Returns a string representation of the regular expression.	16
		Auxiliary methods	938
Total:			1,114

Table 5.3: Implemented methods of the ES5 RegExp object.

5.4 ES5 JSON

Internal representation In Figure 2.8, presented in Section 2.5, we analysed the ES5 JSON object's graph, whose implementation in ECMA-SL is represented by Figure 5.4. Our ECMA-SL implementation of the JSON object graph coincides almost exactly with the one discussed in Section 2.5, except that the named properties of the JSON object are stored in a separate object whose location is stored in its property `JSProperties`. Hence, the methods exposed by the JSON object, `stringify` and `parse`, are stored as properties of the object `JSON.JSProperties`.

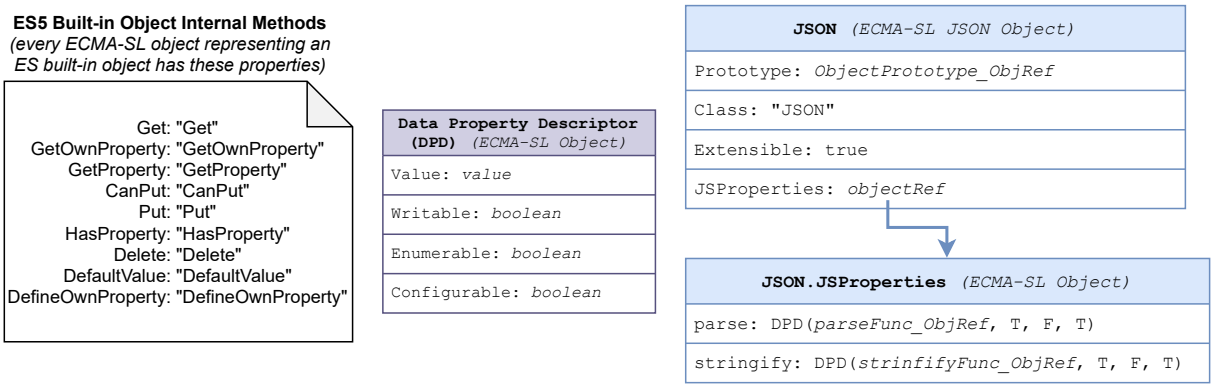


Figure 5.4: ES5 JSON Object Graph in ECMA-SL.

Line-by-line closeness Listing 5.6 shows the implementation of the JSON's `parse` method in ECMA-SL, whose pseudo-code was previously shown in Figure 2.10 of Section 2. Some instructions of the pseudo-code leave the implementation algorithm to the discretion of the programmer. These instructions cannot therefore follow a line-by-line closeness approach.

```
1  function jsonParse(global, this, strict, args) {
2    text := l_nth(args, 0);
3    reviver := getOptionalParam(args, 1);
4    JText := ToString(text);
5    /* 2. Parse JText using the grammars in 15.12.1. Throw a SyntaxError
      exception if JText did not conform to the JSON grammar for the goal
      symbol JSONText. */
6    objJSON := parseJSONText(global, this, strict, [JText]);
7    unfiltered := objJSON;
8    if (IsCallable(reviver) = true) {
9      root := ObjectConstructor(global, 'null, strict, [null]);
10     descriptor := newDataPropertyDescriptorFull(unfiltered, true, true,
       true);
11     {root.DefineOwnProperty}(root, "", descriptor, false);
12     return Walk(root, "", reviver)
13   }
14   else {
15     return unfiltered
16   }
17 };
```

Listing 5.6: JSON's `parse` method in ECMA-SL.

In this implementation, we can see that the method `parseJSONText`, which we have named and defined ourselves, is called in line 6. Unlike the rest of the code of the method `parse`, for which the ECMA-262 standard tells us the exact steps that must be taken, here the standard simply says that

the provided string is to be parsed according to a given set of grammar rules. Hence, we were free to implement this function as we deemed appropriate. We considered two main options: implement the function in OCaml and create an ECMA-SL operator exposing the OCaml function to the ECMA-SL code, or implement the function directly in ECMA-SL. We chose to go for the second option as it is much easier to manipulate and interact with ECMA-SL objects in ECMA-SL code than at the OCaml level. In order to parse the JSON text, we made use of several auxiliary functions, which are presented in Table 5.4.

Importantly, pseudo-code instructions that leave the implementation to the discretion of the programmer raise a problem when it comes to the generation of the English HTML description of ECMA-262, as we cannot generate the original text from it. In order to solve this problem, ECMA-SL uses annotations. However, due to time constraints, we have left this topic outside the scope of this thesis.

Implemented methods Table 5.4 shows the methods related to the ES5 JSON object that we implemented in ECMA-SL as part of this work. For each method, a brief description is given along with the lines of code (LOC) required to implement it. The ES5 JSON object was fully implemented.

Section	Name	Description	LOC
		Set up JSON's Object Graph	
15.12	`initJsonObject`	Creates the JSON object as illustrated by Figure 5.4.	17
		Methods of the JSON object	
15.12.2	`parse`	Parses a JSON-formatted String and produces an ES value. It also receives an optional *reviver* parameter that can filter and transform the results, with the help of the `Walk` abstract method.	16
15.12.3	`stringify`	Receives an ES value and returns the corresponding JSON-formatted String. An optional *replacer* parameter can be passed, that can be either a function that alters the stringification process of objects and arrays, or an array of String and Number objects that white-lists the object properties to stringify. And there can be a *space* parameter to improve human-readability. This method makes use of the `Str`, `Quote`, `JO`, and `JA` abstract methods.	82
		Abstract methods	
15.12.2	`Walk`	A recursive method that is called for each element of a JSON array and each property of a JSON object.	38
15.12.3	`Str`	Turns an ES value into a JSON string. This method makes use of the `Quote`, `JO`, and `JA` abstract methods.	51
15.12.3	`Quote`	Used to wrap a String in double quotes and escape certain characters within it. Here we make an adjustment to have the same behaviour as the 10th edition of ECMA-262 [59], where Unicode surrogate characters are also escaped.	40
15.12.3	`JO`	Serializes an object, calling `Str` for each property of the object.	48
15.12.3	`JA`	Serializes an Array, calling `Str` for each property of the Array.	38

Auxiliary methods			
	`getSortedPropertiesES6`	In order to comply with section 9.1.12 of the ES6 standard [2], we get the property names with positive integer keys in ascending order first, followed by the remaining properties in ascending creation order.	19
	`getOwnEnumerablePropertiesNames`	Gets the enumerable properties of an object, sorted by `getSortedPropertiesES6`. This method is used by both `parse` and `stringify` when iterating over an object's properties.	16
	`concatenateListStrElmsWithSeparator`	Used by the `JO` and `JA` abstracted methods. Concatenates a list of strings using a given *separator* argument.	17
	`isDigit`	Receives a character and verifies if it is a digit. Used by `getTokens`.	4
	`unescapeJSONString`	Receives a string and converts the escapes within it to the characters they represent. Used by `getTokens`.	41
	`getTokens`	Receives a JSON-formatted string and returns a list with tokens. Throws an exception if the string is not valid JSON.	64
	`parseJSONText`	Parses a JSON-formatted string according to ECMA-404 and returns the corresponding ES value in ECMA-SL. Uses `getTokens` and `parseJSONValue`.	14
	`parseJSONValue`	Converts a token from a JSON-formatted string to the corresponding ES value in ECMA-SL.	31
	`parseJSONObject`	Creates an glsES object in ECMA-SL from the list of tokens of the corresponding JSON-formatted string.	41
	`parseJSONArray`	Creates an glsES Array object in ECMA-SL from the list of tokens of the corresponding JSON-formatted string.	31
	`findClosingCurlyBracket`	Receives a list of tokens of a JSON-formatted string, starting at the opening of a JSON object, and returns a list of tokens until the closing bracket for this object is found.	17
	`findClosingSquareBracket`	Receives a list of tokens of a JSON-formatted string, starting at the opening of a JSON array, and returns a list of tokens until the closing bracket for this array is found.	17
	`isUnicodeSurrogate`	Verifies if a given character is a Unicode leading surrogate character. Used by `Quote`.	9
Total:			651

Table 5.4: Implemented methods of the ES5 JSON object.

5.5 ES6 Promise

Internal representation In Figure 2.11, presented in Section 2.6, we had the opportunity to analyse the ES6 Promise object's graph, whose corresponding implementation in ECMA-SL is represented by Figure 5.5.

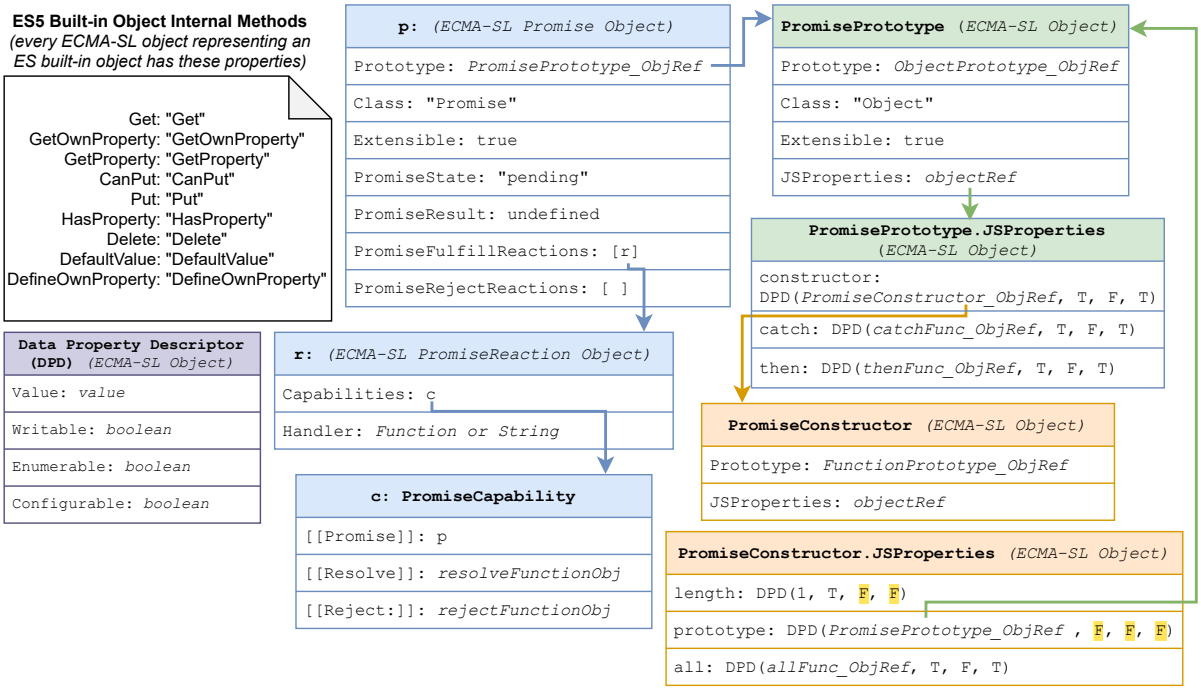


Figure 5.5: ES6 Promise Object Graph in ECMA-SL.

Our implementation of the ES6 Promise built-in object was made on top of `ECMARef5`, in parallel with the transitioning of `ECMARef5` to `ECMARef6`. For this reason, the ES5 built-in object internal methods were kept, along with the `[[Class]]` internal property. The Symbol keyed properties were not implemented.

Line-by-line closeness The implementation of the ES6 Promise built-in object, unlike the implementation of the ES5 built-in objects presented heretofore, was generated entirely by the `HTML2ECMA-SL` tool, which will be introduced in Chapter 6, with the exception of the ECMA-SL functions responsible for creating the Promise object's graph and other auxiliary functions. Listing 5.7 and Listing 5.8 present the ECMA-SL code for the pseudo-code previously shown in Figure 2.12. The code presented in both Listings was fully generated by `HTML2ECMA-SL`.

```
1 function FulfillPromise(promise, value) {
2   assert(promise.PromiseState = "pending");
3   reactions := promise.PromiseFulfillReactions;
4   promise.PromiseResult := value;
5   promise.PromiseFulfillReactions := 'undefined;
6   promise.PromiseRejectReactions := 'undefined;
7   promise.PromiseState := "fulfilled";
8   return TriggerPromiseReactions(reactions, value)
9 };
```

Listing 5.7: An ECMA-SL implementation of the `FulfillPromise` abstract operation.

```
1 function TriggerPromiseReactions(reactions, argument) {
2   foreach (reaction : reactions) {
3     EnqueueJob("PromiseJobs", "PromiseReactionJob", [reaction, argument
    ])
4   };
5   return 'undefined
6 };
```

Listing 5.8: An ECMA-SL implementation of the `TriggerPromiseReactions` abstract operation.

ES6 Dependencies In order to implement the ES6 Promise built-in object, we also had to make a temporary implementation of Jobs and Job Queues, specified in Section 8.4 of the ECMA-262 6th Edition [2], so that the Promise object could be tested against Test262 [3]. Other ES6 internal functions that are used by the Promise object, such as operations on iterator objects, were implemented by other contributors to the ECMA-SL project.

Implemented methods Table 5.5 shows the methods related to the ES6 Promise object that were implemented in ECMA-SL as part of this work. For each method a brief description is given along with the lines of code (LOC) required to implement it. The ES6 Promise object was fully implemented.

Section	Name	Description	LOC
		Set up Promise's Object Graph	
25.4.4	`initPromiseObject`	Creates the Promise constructor object and calls `initPromisePrototype` to create the Promise prototype object. Returns the Promise constructor object	21
25.4.5	`initPromisePrototype`	Creates the Promise prototype object, setting all of its internal and named properties.	14
		Promise constructor	
25.4.3.1	`PromiseConstructor`	Receives an executor function, which captures the computation to be performed asynchronously, and constructs a Promise.	22
	`InternalPromiseConstructor`	An ES5 alternative to `OrdinaryCreateFromConstructor`, used by `PromiseConstructor`, which sets the internal properties `[[Prototype]]`, `[[Extensible]]`, and `[[Class]]` of the new Promise object.	13
		Methods of the Promise constructor	
25.4.4.1	`all`	Receives an iterable of Promise objects, such as an Array, and returns a new Promise object, which is either fulfilled with an Array of the fulfilment values for the passed promises, or rejected with the reason of the first Promise that rejects.	25
25.4.4.3	`race`	Receives an iterable of Promise objects, such as an Array, and returns a Promise that is settled with the same value of the first Promise that is settled in the iterable.	25
25.4.4.4	`reject`	Returns a Promise object that is rejected with a given reason.	12

25.4.4.5	`resolve`	Returns a Promise object that is resolved with a given value. If the value given is a Promise, then that Promise is returned.	19
		Methods of the Promise prototype	
25.4.5.1	`catch`	Equivalent to calling `then`, but only with a function to execute when the Promise is rejected.	5
25.4.5.3	`then`	Called on a Promise object, optionally receives: a function to run when the promise is fulfilled, which receives the fulfillement value; and a function to run when the promise is rejected, which receives the rejection reason. These functions are executed asynchronously.	13
		Non-Promise Abstract methods	
6.2.2.1	`NormalCompletion`	Creates a new Completion record with a given value.	3
6.2.2.1	`Completion`	Equivalent to `NormalCompletion`.	3
8.4.1	`EnqueueJob`	Adds a Job to a Job Queue. Our implementation of this abstract operation is incomplete, having only the purpose of allowing us to test the Promise object.	7
8.4.2	`NextJob`	Unimplemented. Place-holder for a future implementation.	3
		Promise Abstract methods	
25.4.1.1.1	`IfAbruptRejectPromise`	A macro, short hand for a sequence of algorithm steps that use a PromiseCapability record.	10
25.4.1.3	`CreateResolvingFunctions`	Creates the `resolve` and `reject` Function objects for a given Promise object.	18
25.4.1.3.1	`PromiseRejectFunctions`	Algorithm to run when a Promise is rejected.	11
25.4.1.3.2	`PromiseResolveFunctions`	Algorithm to run when a Promise is resolved.	27
25.4.1.4	`FulfillPromise`	Called by `PromiseResolveFunctions`. Changes a Promise's state from "pending" to "fulfilled", sets its PromiseResult internal property, clears the internal reactions lists and calls `TriggerPromiseReactions`.	9
25.4.1.5	`NewPromiseCapability`	Creates a new PromiseCapability using a constructor function that supports the parameter conventions of the Promise constructor.	22
25.4.1.5.1	`GetCapabilitiesExecutorFunctions`	Called by `NewPromiseCapability` to set the executor function of the new Promise object of the PromiseCapability.	15
25.4.1.6	`IsPromise`	Checks if a value is a Promise object.	9

25.4.1.7	`RejectPromise`	Called by `PromiseRejectFunctions` and `PromiseResolveFunctions`. Changes a Promise's state from "pending" to "rejected", sets its PromiseResult internal property, clears the internal reactions lists and calls `TriggerPromiseReactions`.	9
25.4.1.8	`TriggerPromiseReactions`	Receives a list of PromiseReaction records and enqueues a new Job for each record.	6
25.4.4.1.1	`PerformPromiseAll`	Used by the `all` method of the Promise constructor in order to get the new Promise.	43
25.4.4.1.2	`PromiseAllResolveElementFunctions`	Resolved a specific `all` element.	19
25.4.4.3.1	`PerformPromiseRace`	Used by the `race` method of the Promise constructor in order to get the first Promise that settles.	23
25.4.5.3.1	`PerformPromiseThen`	Used by the `then` method of the Promise prototype in order to schedule either the function to execute when the Promise is resolved, or the function to execute when the Promise is rejected.	25
		Promise Jobs	
25.4.2.1	`PromiseReactionJob`	Receives a PromiseReaction and an argument and either resolves or rejects the associated Promise object.	21
25.4.2.2	`PromiseResolveThenableJob`	Calls the `then` method associated with the Promise object.	10
		Auxiliary methods	
	`newPendingJob`	Creates a new Job object, which contains the name of the function to execute and a list of arguments.	22
	`initJobQueue`	Creates a list global variable, `JobQueue`. This function is called by the interpreter before a program's top level execution.	4
	`appendToJobQueue`	Adds a Job to `JobQueue`.	4
	`executeJobs`	Executes the Jobs in the `JobQueue`, calling the abstract operations associated with each Job.	22
	`getPromiseConstructor`	Returns the Promise constructor object.	4
	`getPromisePrototype`	Returns the Promise prototype object.	5
	`getPromiseConstructorFromPromiseObject`	Returns the Promise constructor object associated with a method of the Promise constructor. Workaround for getting the `@@species` property.	7
6.2.2	`isCompletionRecord`	Checks if a value is a Completion record.	3

25.4.1.1	`isPromiseCapabilityRecord`	Checks if an object is a PromiseCapability record.	3
25.4.1.2	`newPromiseReaction`	Creates a new PromiseReaction record.	3
25.4.1.2	`isPromiseReactionRecord`	Checks if an object is a PromiseReaction record.	3
Total:			542

Table 5.5: Implemented methods of the ES6 Promise object.

Chapter 6

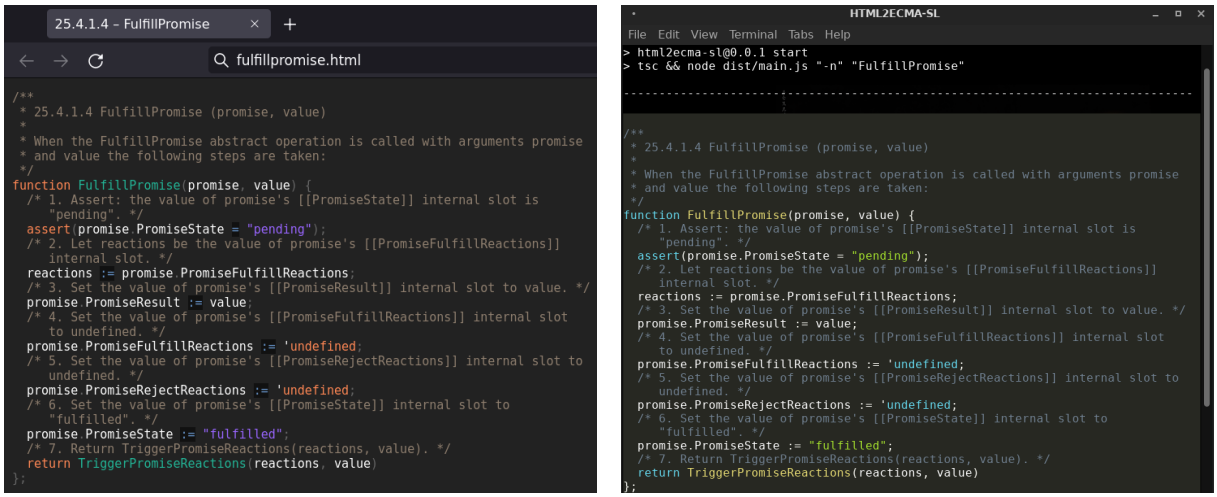
HTML2ECMA-SL

During the implementation of our first ES built-in object, the ES5 Array, we came to the realization that this task was considerably repetitive. For each method, we would always perform the same steps:

1. Copy the method's section number, name, parameters, summary, pseudo-code, and footer notes from the HTML version of ECMA-262 to our ECMA-SL source file. Comment out each pseudo-code instruction individually and the remaining information about the method.
2. In the HTML version of ECMA-262, the pseudo-code instructions of a method are structured in an HTML ordered list, whose number (or letter) prefixes cannot be selected in the browser. For this motive, we add them manually to each individual comment of a pseudo-code instruction.
3. Insert new lines in the comments so that each line does not exceed 80 characters, for readability.
4. Add the method's signature and add curly braces around the pseudo-code comments.
5. Implement each pseudo-code instruction below its respective comment. Search for equivalent implementations in our reference interpreter, in order to keep consistency.
6. Make sure that the indentation of a block is 2 spaces.

This process can become quite dull and error-prone, which led us to develop `HTML2ECMA-SL` for the purpose of automating it. `HTML2ECMA-SL` searches for a function in the HTML version of the ECMA-262 6th Edition [10] and parses the HTML description of the function's pseudo-code into functional ECMA-SL code, by taking advantage of regular expressions. Figure 6.1 illustrates our results for the `FulfillPromise` method described in Section 25.4.1.4 of the ECMA-262 6th Edition [10], whose pseudo-code had been previously shown in Figure 2.12. `HTML2ECMA-SL` allows us to output the generated ECMA-SL code of a function to: the console, with or without syntax highlighting, which is useful for debugging and copy/pasting; an ECMA-SL source file, which will become useful when `HTML2ECMA-SL` is capable of generating a whole section of the standard; and an HTML file with syntax highlighting, which may be useful for embedding the generated ECMA-SL code in a web application or PDF document. On top of this, `HTML2ECMA-SL` also allows us to specify the number of spaces or tabs to use for the indentation, the maximum line width of the comments, and whether to generate the comments or not.

We have fully implemented the automatic generation of ECMA-SL code for the ES6 Promise and ES6 Proxy built-in objects. Because many pseudo-code instructions have recurring patterns, many other methods are already partially or fully implemented, including those belonging to newer versions of ECMA-262, although these require a few modifications to our existing regular expressions. This means that, even if we cannot generate ECMA-SL code for the entirety of ECMA-262, we can already automate



(a) HTML output.

(b) Console output.

Figure 6.1: The `FulfillPromise` method generated by `HTML2ECMA-SL`.

many of the repetitive tasks, namely, the generation of comments and the generation of ECMA-SL instructions whose patterns have already been implemented in `HTML2ECMA-SL`. At this stage, six other contributors to the ECMA-SL project are already enjoying the benefits of this tool.

Project structure We decided to develop `HTML2ECMA-SL` in TypeScript (TS), a superset of JS that adds a type system to the language. For the runtime we use Node.js [1]. Because the focus of ECMA-SL is on ES, this can help us further understand modern ES from a user's perspective. The most relevant source code of `HTML2ECMA-SL` is divided into the following ES modules [2]:

- `main.ts` - This is the entry point of the application, responsible for: parsing the given command line arguments, using the Yargs [3] Node.js library for this purpose, which allows us to specify various options on how to output the generated ECMA-SL code; fetching the HTML version of the ECMA-262 6th Edition [10] and extracting the HTML portion of a given function's specification from it, using the `ecma262.ts` module; and parsing the HTML of the function's specification and generating the corresponding ECMA-SL code, using the `parser.ts` module.
- `ecma262.ts` - Responsible for: fetching the HTML of the ECMA-262 6th Edition [10] from its URL, caching it to a local file for future uses; fetching the HTML portion of a function's specification from the HTML of ECMA-262; and verifying if a given function is a constructor function by searching for its name in Section 18.3 of the ECMA-262 6th Edition.
- `parser.ts` - Responsible for parsing the HTML portion of a function's specification and producing the corresponding ECMA-SL source code, taking into account the indentation, line-width, and comment generation preferences.
- `syntax_highlighting.ts` - Adds syntax highlighting to the ECMA-SL source code by using the Prism [4] syntax highlighting library, which generates an HTML output that can then be converted to Linux console or Windows Terminal [5] syntax-highlighted text by using the Chalk [6] Node.js library.

[1] Node.js, 31 October 2021 - `https://nodejs.org/en/`
[2] ES modules were introduced in the ECMA-262 6th Edition [10] and allow us to split our code in multiple files.
[3] Yargs, 31 October 2021 - `https://www.npmjs.com/package/yargs`
[4] Prism, 31 October 2021 - `https://www.npmjs.com/package/prismjs`
[5] Windows Terminal, 31 October 2021 - `https://github.com/microsoft/terminal`
[6] Chalk, 31 October 2021 - `https://www.npmjs.com/package/chalk`

Aside from the aforementioned modules, we also have a test directory containing TS source files for unit tests, in which we use the Jest [9] testing framework. In these tests, we verify if the text of a generated ECMA-SL function, whose instructions we have implemented, is as expected. Currently, HTML2ECMA-SL consists of 1,472 LOC for the source code and 1,188 LOC for the unit tests.

Parsing instructions The general structure of the HTML portion of an ECMA-262 function is illustrated in Listing 6.1, featuring the HTML SECTION tag of the specification of the FulfillPromise internal method. Most HTML attributes are irrelevant to our parser, and therefore were omitted.

```
<section ...>
  <h1><span ...><a ...>25.4.1.4</a></span> FulfillPromise ( promise, value)</h1>
  <p ...>When the FulfillPromise abstract operation is called with arguments <var>promise</var> and <var>value</var> the following
     steps are taken:</p>
  <ol class="proc">
    <li><a ...>Assert</a>: the value of <i>promise</i>'s [[PromiseState]] <a ...>internal slot</a> is <code>"pending"</code>.</li>
    <li>Let <i>reactions</i> be the value of <i>promise</i>'s [[PromiseFulfillReactions]] <a ...>internal slot</a>.</li>
    <li>Set the value of <i>promise</i>'s [[PromiseResult]] <a ...>internal slot</a> to <i>value</i>.</li>
    <li>Set the value of <i>promise</i>'s [[PromiseFulfillReactions]] <a ...>internal slot</a> to <b>undefined</b>.</li>
    <li>Set the value of <i>promise</i>'s [[PromiseRejectReactions]] <a ...>internal slot</a> to <b>undefined</b>.</li>
    <li>Set the value of <i>promise</i>'s [[PromiseState]] <a ...>internal slot</a> to <code>"fulfilled"</code>.</li>
    <li>Return <a ...>TriggerPromiseReactions</a>(<i>reactions</i>, <i>value</i>).</li>
  </ol>
</section>
```

Listing 6.1: FulfillPromise's HTML SECTION tag.

The structure of the HTML SECTION tag of a function's specification is divided in three parts: the header, which contains the function's section, name, parameters, and summary; the pseudo-code, which is identified by an OL tag with class "proc"; and the footer, which contains notes and/or miscellaneous information about the function. Some function (or macro) specifications may have more than one main block of pseudo-code, as is the case of the IfAbruptRejectPromise macro (Section 25.4.1.1.1 of ES6).

In order to extract the necessary information from the HTML, we make extensive usage of the RegExp object, discussed in Section 2.4 of this document. This is possible because the patterns of function specifications are generally the same, as well as the patterns of recurring pseudo-code instructions. Listing 6.2 demonstrates the regular expression pattern that we use for extracting the information corresponding to a function's signature. We obtain the function's section identifier with the first capturing group, highlighted in red, the function's name with the second capturing group, highlighted in green, and the function's parameters with the third capturing group, highlighted in blue. The third group requires further text processing in order to extract the individual parameters from the captured string. From the second capturing group, we can not only extract the function's name, but we can also infer if the function is internal or built-in, by verifying if the name contains the dot character, which is used to access properties of an object.

```
const re = /<h1><span[^>]*><a[^>]*>([^<]*)<\/a><\/span>\s*([^(]+)(\([^)]*\))?\s*<\/h1>/;
```

Listing 6.2: RegExp for extracting a function's signature from the HTML.

For parsing the pseudo-code instructions, we parse each HTML LI tag, that is, each instruction, individually. Each instruction is matched against a list of possible regular expression patterns until a match is found. Because instructions may contain several expressions, this process can repeat for some of the capturing groups. As we have not written regular expressions to cover all the possible patterns of pseudo-code instructions in the standard, it is often the case that HTML2ECMA-SL is not able to create an ECMA-SL statement corresponding to the instruction to be parsed. In such cases, the tool simply outputs `/* TODO: Instruction not yet implemented. */` and continues with the generation process.

Chapter 7

Evaluation

In this chapter, we present the evaluation results for the main topics of this work: the implementation of the ES5 Array, part of the ES5 String, ES5 RegExp, and ES5 JSON built-in objects in `ECMARef5`; the implementation of the ES6 Promise built-in object in `ECMARef6`; and `HTML2ECMA-SL`.

7.1 Reference Implementations

At this stage, our evaluation is primarily done with Test262 [3], the official ECMAScript Conformance Test Suite. Despite its known coverage issues, Test262 is the most comprehensive ES test suite to date. At a later stage in the ECMA-SL project, it will be possible to measure the coverage of Test262, with respect to the code of the `ECMARef5` interpreter, in order to know which features have not yet been tested.

Test filtering The evolution of Test262 is consistent with that of ECMA-262. Thus, Test262 contains tests that target the latest edition of the standard—currently ES12—which our ES5 and ES6 reference implementations cannot handle. Furthermore, as of October 2021, the number of tests from Test262 covering ES built-in objects is over 19k. Fortunately, many of the Test262 test files targeting ES5 contain the ”es5id” key in the meta-data of the test, known as the frontmatter [1]. This simplifies some of the filtering process required in our project. Figure 7.1 shows the frontmatter of two Test262 test files that illustrate this, for the ES5 Array and for the ES6 Promise built-in objects, respectively.

```
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]', 's');
```

(a) 15.4.5-1.js

```
1  // Copyright (C) 2015 the V8 project authors. All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  es6id: 25.4.3.1
6  description: Promise executor is invoked synchronously
7  info: |
8      9. Let completion be Call(executor, undefined,
9         «resolvingFunctions.[[Resolve]], resolvingFunctions.[[Reject]]»).
10
11     25.4.1.3.2 Promise Resolve Functions
12
```

(b) exec-args.js

Figure 7.1: Meta-data of two Test262 test files.

Unfortunately, both the ”es5id” and ”es6id” keys have been deprecated [2]. New tests contain a ”esid” key, without the version number, whose value is the hash ID of the HTML anchor of a section of the

[1] Test262 frontmatter, 31 October 2021 - `https://github.com/tc39/test262/blob/main/CONTRIBUTING.md#frontmatter`

[2] Test262 Technical Rationale Report, 31 October 2021 - `https://github.com/tc39/test262/wiki/Test262-Technical-Rationale-Report,-October-2017#specification-reference-ids`

latest ECMA-262's HTML version. For this reason, many tests with the "esid" key may be supported by ES5 and/or ES6. We filter these tests manually as we investigate the causes for the failing tests during development. In some cases, there are small incompatibilities between different versions of ES, such as the "length" property of Function objects being configurable from ES6 onwards. For these cases, we have chosen to adapt the behaviour of our reference interpreter to the latest version of ECMA-262.

Testing pipeline Test262 test files make use of auxiliary functions for run-time assertions. For instance, the `assert.sameValue` function, shown in line 12 of Figure 7.1 (a), compares the values of the first two arguments, and throws a custom error, `Test262Error`, if they differ, causing the test to fail. The set of auxiliary functions used in test files comes from a collection of helper files called the *harness* of Test262. In order to run a test file with the ECMA-SL project we must first include the harness, which we do by concatenating it with the test file prior to running it. The frontmatter of a test file gives us a few instructions on how the test should be executed and what to expect of it. For instance, if the frontmatter contains the key `flags`, and if this key contains the boolean value `onlyStrict` in its list of values, then we must append the `"use strict"` directive to the code of the test, which activates strict mode. Also, if the frontmatter of a test file contains the key `negative`, then the test is expected to throw an exception, whose error name is specified in the string value associated with this key. If the key `negative` is absent, then any error thrown will cause the test to fail.

In order to automate the evaluation of our reference implementation we make use of a shell script that automates the testing process. This process is described in Figure 7.2 and works as follows: (1) we check if the frontmatter of the test includes the boolean value `onlyStrict` in the list of boolean values referenced by the key `flags`, in which case we prepend the directive `"use strict"` directive to the code of the test; (2) we prepend the code of the Test262 harness to the code of the test; (3) we compile the test to ECMA-SL and execute it using the ECMA-SL engine; (4) we analyse the result of executing the test, as discussed above, negative tests are expected to throw an exception of a certain type, while positive tests are expected to execute normally.

After executing the pipeline for a given list of tests, we are given the information of which tests have passed and which tests have failed, as well as the total passing and total failing tests.

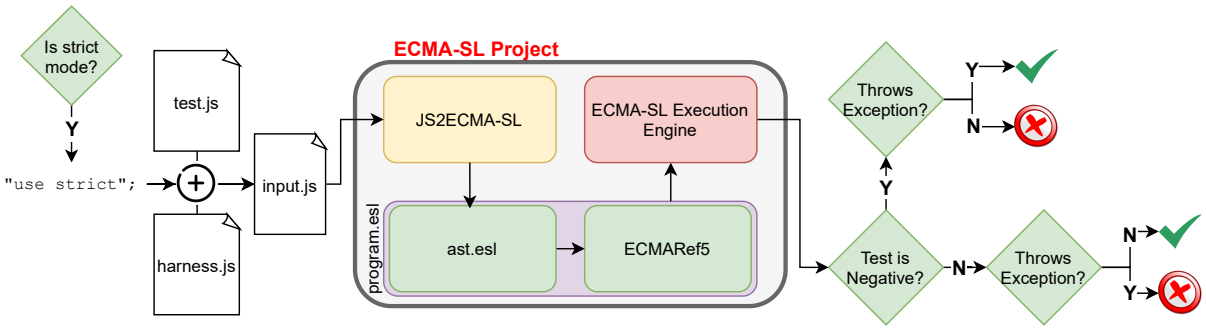


Figure 7.2: Test execution pipeline.

Test results The test results of our reference implementation for the `ECMARef5` interpreter are presented in detail in the tables of Appendix B, and summarized in Table 7.1. In total, our ES5 reference implementation passes 3,433 out of the 3,440 filtered tests, with 7 tests failing. Many dependencies of the ES6 Promise built-in object, such as syntactic elements and built-in functions, were not implemented before the conclusion of this work. Thus, only 171 of a total of 613 filtered tests for the ES6 Promise object are currently passing.

With respect to the failing tests, we currently have three tests failing for the methods that we implemented for the ES5 String object, which are caused by a faulty conversion of large numbers to string values in ECMA-SL. This is because the current implementation of the `ToString` method applied to the Number type, as specified in Section 9.8.1 of the ECMA-262 Edition 5.1 [5], is incomplete in `ECMARef5`. The test that currently fails for the JSON object has to do with the attempt of stringifying Unicode surrogate values. A Unicode surrogate value is currently replaced with the $FFFD_{16}$ code point by `JS2ECMA-SL`, as long as we are using UTF-8, which causes the stringify operation to generate a wrong result. Finally, for the RegExp object, we have two tests failing due to backreferences appearing before their capture groups being treated as decimal escapes by the `regexp-tree` library [3], which we use for parsing the string representation of an ES regular expression pattern to its corresponding AST. The remaining test that is currently failing for the RegExp object is due to an implementation error in the handling of backslash characters in the string representation of RegExp patterns.

Section	Description	#T	Passed	Failed
`15.4`	`Array`	117	117	0
`15.4`	`Array prototype`	2150	2150	0
`15.5.4`	`String prototype`	538	535	3
`15.12`	`JSON`	116	115	1
`15.10`	`RegExp`	109	109	0
`15.10`	`RegExp grammar`	292	289	3
`15.10`	`RegExp prototype`	118	118	0
Total		3,440	3,433	7

Table 7.1: Test262 test results for our ES5 reference implementation.

[3] Backreferences appearing before their capture groups incorrectly treated as decimal escapes, 31 October 2021 - `https://github.com/DmitrySoshnikov/regexp-tree/issues/69`

7.2 HTML2ECMASL

The evaluation of HTML2ECMA-SL is a challenging task, since this application is not expected to produce fully functional ECMA-SL code right out-of-the-box, considering that the ECMA-SL project is still under development. It will often be necessary to review the generated code, write auxiliary functions, and, in some cases, extend ECMA-SL with new operators and syntax. Hence, we cannot simply evaluate HTML2ECMA-SL by having it generate several sections of the ECMA-262 standard and then check if the generated code passes the corresponding Test262 tests.

In order to evaluate HTML2ECMA-SL, we apply it to Sections 25 and 26 of the ECMA-262 6th Edition [10], and check whether the generated ECMA-SL code is syntactically valid or not. These sections correspond to the Promise and Proxy built-in objects, respectively. To automate the testing process, we make use of unit tests, resorting to the Jest [9] testing framework. Each unit test compares the generated ECMA-SL code for a given function described in the ECMA-262 standard with the code that we expect to be generated, which is susceptible to change over time as we adjust it to work with ECMARef6.

As previously mentioned, we could not have fully functional code for the ES6 Promise built-in object due to the lack of syntactic and built-in function elements in ECMARef6, which is currently under development. We also did not test the code generated for the ES6 Proxy built-in object in ECMARef6. Despite this, testing that the ECMA-SL code is generated as expected is important in order to ensure that future changes to HTML2ECMA-SL do not introduce silent bugs and corrupt the behaviour of the application. HTML2ECMA-SL currently passes 51 out of 51 unit tests, with a test coverage of 97%, measured by Codecov [4].

[4] Codecov is a dedicated code coverage solution, which we use to measure the coverage of our tests, 31 October 2021: https://about.codecov.io

Chapter 8

Conclusion

With the steady evolution of the ECMA-262 standard, it becomes increasingly difficult to maintain and extend this language specification. For this reason, we contribute to the ECMA-SL project by implementing the following built-in objects in its ES5 reference interpreter: Array, String (partial), RegExp, and JSON. This work paves the way for the implementation of these built-in objects in future ES reference interpreters, written in ECMA-SL, that target later versions of ECMA-262. Additionally, we further contribute to the ECMA-SL project by devising a tool, which we have called `HTML2ECMA-SL`, that considerably simplifies the implementation process by automatically generating ECMA-SL code from the pseudo-code of the ECMA-262 6th Edition [10]. We were able to fully generate the ECMA-SL code for the ES6 Promise and Proxy objects, which we used to initiate the development of an ES6 reference interpreter written in ECMA-SL.

The three main topics of this thesis were thoroughly evaluated. Our reference implementation of the ES5 Array, String, RegExp, and JSON built-in objects was tested against Test262 [3], passing 3,433 out of the 3,440 filtered tests. At this stage, we do not yet measure the coverage of Test262, nor apply formal methods to the language. Our reference implementation of the ES6 Promise built-in object was also tested against Test262, passing 171 out of 613 filtered tests. And, finally, `HTML2ECMA-SL` was tested using a custom made test suite consisting of 51 unit tests, each targeting a specific algorithm / function of the ECMA-262 standard. The proposed test suite has a 97% code coverage of `HTML2ECMA-SL`, giving us a strong guarantee of its correctness.

Our contribution to the `ECMARef5` interpreter has a total of 3,184 LOC, our contribution to the `ECMARef6` interpreter has a total of 542 LOC, and `HTML2ECMA-SL` has a total of 2,660 LOC. The ECMA-SL project, as well as `HTML2ECMA-SL`, will become open-source in the near future.

Future work We categorize the future work in two types: immediate and long-term. Due to the time constraints of this project, we were unable to apply `HTML2ECMA-SL` to the entirety of the ECMA-262 6th Edition [10] and to generate a given built-in object's graph in ECMA-SL from the specification of the standard. Hence, our immediate future work would be to extend `HTML2ECMA-SL` in order to recognize more patterns of pseudo-code instructions occurring in the ES6 specification, as well as to generate a given built-in object's graph in ECMA-SL from its specification in the standard. Our implementation of the ES6 Promise object in the `ECMARef6` interpreter is also failing a large percentage of tests, which we would seek to correct by extending the ECMA-SL language and the `ECMARef6` interpreter as necessary. In the long-term, we would like to adapt the patterns of pseudo-code instructions that occur in the ECMA-262 6th Edition [10] to the later editions.

Bibliography

[1] C. Severance, "Javascript: Designing a language in 10 days," *Computer*, vol. 45, pp. 7–8, 02 2012.

[2] "Ecmascript® language specification, 6^{th} edition / june 2015." `https://www.ecma-international.org/ecma-262/6.0`. Accessed on 2021-10-31.

[3] "Test262 - official ecmascript conformance test suite." `https://github.com/tc39/test262/`. Accessed on 2021-10-31.

[4] S. Anand, E. Burke, T. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. J. Harrold, A. Bertolino, J. Li, and H. Zhu, "An orchestrated survey on automated software test case generation i," 2013.

[5] "Annotated, hyperlinked, html version of edition 5.1 of the ecmascript specification." `https://es5.github.io/`. Accessed on 2021-10-31.

[6] "Ocaml - general-purpose, multi-paradigm programming language." `https://ocaml.org/`. Accessed on 2021-10-31.

[7] "Menhir - lr(1) parser generator for the ocaml programming language." `http://gallium.inria.fr/~fpottier/menhir/`. Accessed on 2021-10-31.

[8] "Ounit - a unit test framework for ocaml." `https://github.com/gildor478/ounit/`. Accessed on 2021-10-31.

[9] "Jest - a javascript testing framework designed to ensure correctness of any javascript codebase." `https://jestjs.io/`. Accessed on 2021-10-31.

[10] "Html rendering of ecma-262 6^{th} edition, the ecmascript 2015 language specification." `https://www.ecma-international.org/ecma-262/6.0/`. Accessed on 2021-10-31.

[11] J. Fragoso Santos, P. Maksimović, D. Naudziuniene, T. Wood, and P. Gardner, "Javert: Javascript verification toolchain," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–33, 12 2017.

[12] J. Mate, "Javascript does not need a stringbuilder." `https://josephmate.github.io/java/javascript/stringbuilder/2020/07/27/javascript-does-not-need-stringbuilder.html`, 7 2020.

[13] "Standard ecma-404, the json data interchange syntax, 2^{nd} edition / december 2015." `https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf`. Accessed on 2021-10-31.

[14] K. Gallaba, A. Mesbah, and I. Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in javascript," pp. 1–10, 10 2015.

[15] P. Thiemann, “Towards a type system for analyzing javascript programs,” *Lecture Notes in Computer Science*, vol. 3444, 04 2005.

[16] C. Anderson, S. Drossopoulou, and P. Giannini, “Towards type inference for javascript,” *Lecture Notes in Computer Science*, vol. 3586, 07 2005.

[17] D. Jang and K.-M. Choe, “Points-to analysis for javascript,” *Proceedings of the ACM Symposium on Applied Computing*, pp. 1930–1937, 01 2009.

[18] C. Park and S. Ryu, “Scalable and precise static analysis of javascript applications via loop-sensitivity,” 01 2015.

[19] K. Dewey, V. Kashyap, and B. Hardekopf, “A parallel abstract interpreter for javascript,” pp. 34–45, 02 2015.

[20] A. Chaudhuri, “Flow: Abstract interpretation of javascript for type checking and beyond,” 10 2016.

[21] J. Fragoso Santos, T. P. Jensen, T. Rezk, and A. Schmitt, “Hybrid typing of secure information flow in a javascript-like language,” 01 2016.

[22] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “Jsflow: Tracking information flow in javascript and its apis,” 03 2014.

[23] A. Chudnov and D. A. Naumann, “Inlined information flow monitoring for javascript,” 10 2015.

[24] P. Gardner, S. Maffeis, and G. Smith, “Towards a program logic for javascript,” *Sigplan Notices - SIGPLAN*, vol. 47, pp. 31–44, 01 2012.

[25] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, “Javert 2.0: compositional symbolic execution for javascript,” *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, 01 2019.

[26] “Coq - interactive formal proof management system.” `https://coq.inria.fr/`. Accessed on 2021-10-31.

[27] “K - rewrite-based executable semantic framework.” `https://kframework.org/`. Accessed on 2021-10-31.

[28] S. Maffeis, J. Mitchell, and A. Taly, “An operational semantics for javascript,” pp. 307–325, 12 2008.

[29] G. Plotkin, “A structural approach to operational semantics,” *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 07 2004.

[30] S. Maffeis and A. Taly, “Language-based isolation of untrusted javascript,” pp. 77–91, 07 2009.

[31] S. Maffeis, J. Mitchell, and A. Taly, “Isolating javascript with filters, rewriting, and wrappers,” vol. 5789, pp. 505–522, 09 2009.

[32] A. Guha, C. Saftoiu, and S. Krishnamurthi, “The essence of javascript,” *ECOOP, Lec- Ture Notes in Computer Science*, pp. 126–150, 06 2010.

[33] “Racket - general-purpose programming language.” `https://racket-lang.org/`. Accessed on 2021-10-31.

[34] J. Politz, M. Carroll, B. Lerner, J. Pombrio, and S. Krishnamurthi, “A tested semantics for getters, setters, and eval in javascript,” *ACM SIGPLAN Notices*, vol. 48, pp. 1–16, 10 2012.

[35] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. 49, pp. 87–100, 01 2014.

[36] A. Charguéraud, "Pretty-big-step semantics," pp. 41–60, 03 2013.

[37] P. Gardner, G. Smith, C. Watt, and T. Wood, "A trusted mechanised specification of javascript: One year on," vol. 9206, pp. 3–10, 07 2015.

[38] "V8 - google's open source high-performance javascript and webassembly engine, written in c++." `https://v8.dev/`. Accessed on 2021-10-31.

[39] D. Park, A. Stefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," *ACM SIGPLAN Notices*, vol. 50, pp. 346–356, 06 2015.

[40] G. Rosu, A. Stefanescu, Ş. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pp. 358–367, 2013.

[41] A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 74–91, 2016.

[42] A. Charguéraud, A. Schmitt, and T. Wood, "Jsexplain: A double debugger for javascript," *WWW '18: Companion Proceedings of the The Web Conference 2018*, pp. 691–699, 04 2018.

[43] J. Launchbury and S. Peyton Jones, "State in haskell," *LISP and Symbolic Computation*, vol. 8, 11 1998.

[44] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1–24, 10 2017.

[45] S. Alimadadi, M. Madsen, D. Zhong, and F. Tip, "Finding broken promises in asynchronous javascript programs," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–26, 10 2018.

[46] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 08 2013.

[47] G. Sampaio, J. F. Santos, P. Maksimović, and P. Gardner, "A trusted infrastructure for symbolic analysis of event-driven web applications," pp. 15–16, 11 2020.

[48] W3C, "Dom core level 1 specification." `https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html`. Accessed on 2021-10-31.

[49] W3C, "Ui events." `https://www.w3.org/TR/uievents/`. Accessed on 2021-10-31.

[50] Wheeler, Ken and Spampinato, Fabio, "cash (github)." `https://github.com/kenwheeler/cash`. Accessed on 2021-10-31.

[51] Sorhus, Sindre, "p-map (github)." `https://github.com/sindresorhus/p-map`. Accessed on 2021-10-31.

[52] J. Park, J. Park, S. An, and S. Ryu, "JISET: javascript ir-based semantics extraction toolchain," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 647–658, 2020.

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

[54] "Esprima - ecmascript parsing infrastructure for multipurpose analysis." `https://esprima.org/`. Accessed on 2021-10-31.

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

[56] M. Goregaokar, "Let's stop ascribing meaning to code points." `https://manishearth.github.io/blog/2017/01/14/stop-ascribing-meaning-to-unicode-code-points/`, 1 2017.

[57] "The unicode character database." `https://www.unicode.org/ucd/`. Accessed on 2021-10-31.

[58] J. Fragoso Santos, L. Almeida, and R. Abreu, "Rexstepper: a reference debugger for javascript regular expressions," submitted.

[59] "Ecmascript® language specification, 10^{th} edition / june 2019." `https://www.ecma-international.org/ecma-262/10.0`. Accessed on 2021-10-31.

Appendix A

Syntax of the ECMA-SL language

Syntax of the ECMA-SL language

Integers: $i \in \mathcal{I}$ Floats: $f \in \mathcal{F}$ Booleans: $b \in \mathcal{B}$

Vars: $x \in \mathcal{X}$ Strings: $s \in \mathcal{S}$ Types: $\tau \in \mathcal{TY}$

Locs: $l \in \mathcal{L}$ Symbol: $sy \in \mathcal{SY}$

Values: $v \in \mathcal{V}$:= $i \mid f \mid s \mid b \mid l \mid \tau \mid [v_1, \cdots, v_n] \mid (v_1, \cdots, v_n) \mid$ void $\mid$ null $\mid sy$

Expressions: $e \in \mathcal{E}$:= $(e) \mid v \mid x \mid |x| \mid \ominus e \mid e_1 \oplus e_2 \mid \otimes(e_1, e_2, e_3) \mid \otimes(e_1, \cdots, e_n) \mid$
$\{\} \mid \{s_1 : e_1, \cdots, s_n : e_n\} \mid e.x \mid e_1[e_2] \mid$
$e(e_1, \cdots, e_n) \mid e(e_1, \cdots, e_n)$ catch $x \mid$ extern $e(e_1, \cdots, e_n)$

Statements: $st \in \mathcal{S}tmts$:= $\{st_1; \cdots; st_n\} \mid$ skip $\mid$ print $e \mid$ fail $e \mid$ assert $e \mid$ throw $e \mid$
if (e) st_1 else $st_2 \mid$ if (e_1) st_1 elif (e_2) st_2 elif $\cdots$ else $st_n \mid$
repeat $st \mid$ repeat st until $e \mid$ while (e) $st \mid$
foreach $(x : e)$ $st \mid$ switch (e) {case $e_1 : st_1 \cdots$ case $e_n : st_n$} $\mid$
match e with $\mid \{s : x \mid v \mid$ None $\} \rightarrow st \mid \cdots \mid$ default $\rightarrow st \mid$
$x :=$ lambda $(x_1, \cdots, x_n)[x_1, \cdots, x_n]$ $st \mid @x(e_1, \cdots, e_n) \mid$
$|x| := e \mid x := e \mid e_1[e_2] := e_3 \mid$ delete $e_1[e_2] \mid$ return e

Syntax of the Core ECMA-SL Language

Integers: $i \in \mathcal{I}$ Floats: $f \in \mathcal{F}$ Booleans: $b \in \mathcal{B}$

Vars: $x \in \mathcal{X}$ Strings: $s \in \mathcal{S}$ Types: $\tau \in \mathcal{TY}$

Locs: $l \in \mathcal{L}$ Symbol: $sy \in \mathcal{SY}$

Values: $v \in \mathcal{V}$:= $i \mid f \mid s \mid b \mid l \mid [v_1, \cdots, v_n] \mid \tau \mid (v_1, \cdots, v_n) \mid$ void $\mid$ null $\mid sy$

Expressions: $e \in \mathcal{E}$:= $v \mid x \mid \ominus e \mid e_1 \oplus e_2 \mid \otimes(e_1, \cdots, e_n)$

Statements: $st \in \mathcal{S}tmts$:= $st_1; st_2 \mid$ skip $\mid$ merge $\mid$ print $e \mid$ fail $e \mid$
if (e) st_1 else $st_2 \mid$ while (e) $st \mid$ return $e \mid$
$x := e \mid x := e(e_1, ..., e_n) \mid x := \{\} \mid x := e_1$ in $e_2 \mid$
$e_1[e_2] := e_3 \mid$ delete $e_1[e_2] \mid x := e_1[e_2] \mid$
$x := e@(e_1, ..., e_n) \mid x :=$ fields e

Appendix B

Reference implementation results

Section (folder)	#T	Passed	Failed
String prototype			
15.5.4.10 (match)	38	38	0
15.5.4.11 (replace)	43	43	0
15.5.4.12 (search)	30	30	0
15.5.4.14 (split)	207	207	0
15.5.4.16 (toLowerCase)	24	24	0
15.5.4.17 (toLocaleLowerCase)	24	24	0
15.5.4.18 (toUpperCase)	23	23	0
15.5.4.19 (toLocaleUpperCase)	23	23	0
15.5.4.20 (trim)	126	123	3
Total	538	535	3

Table B.1: Test262 test results for the methods of the ES5 String object that we implemented.

Section (folder)	#T	Passed	Failed
JSON			
15.12 (JSON)	5	5	0
15.12.2 (parse)	63	63	0
15.12.3 (stringify)	48	47	1
Total	116	115	1

Table B.2: Test262 test results for the ES5 JSON object.

Section (folder)	#T	Passed	Failed
Array			
15.4 (Array)	59	59	0
15.4.2 (constructor)	2	2	0
15.4.3.2 (isArray)	25	25	0
15.4.5.2 (length)	14	14	0
15.4.4 (prototype)	17	17	0
Array prototype			
15.4.4.2 (toString)	10	10	0
15.4.4.3 (toLocaleString)	9	9	0
15.4.4.4 (concat)	13	13	0
15.4.4.5 (join)	21	21	0
15.4.4.6 (pop)	15	15	0
15.4.4.7 (push)	14	14	0
15.4.4.8 (reverse)	14	14	0
15.4.4.9 (shift)	17	17	0
15.4.4.10 (slice)	47	47	0
15.4.4.11 (sort)	27	27	0
15.4.4.12 (splice)	55	55	0
15.4.4.13 (unshift)	15	15	0
15.4.4.14 (indexOf)	190	190	0
15.4.4.15 (lastIndexOf)	191	191	0
15.4.4.16 (every)	207	207	0
15.4.4.17 (some)	208	208	0
15.4.4.18 (forEach)	183	183	0
15.4.4.19 (map)	187	187	0
15.4.4.20 (filter)	220	220	0
15.4.4.21 (reduce)	254	254	0
15.4.4.22 (reduceRight)	253	253	0
Total	2,267	2,267	0

Table B.3: Test262 test results for the ES5 Array object.

Section (folder)	#T	Passed	Failed
RegExp			
15.10 (RegExp)	11	11	0
15.10.1 (matcher-syntax)	16	16	0
15.10.3.1 (regexp-object)	9	9	0
15.10.4.1 (new-regexp)	47	47	0
15.10.5 (constructor-properties)	3	3	0
15.10.5.1 (constructor)	4	4	0
15.10.7 (instance-properties)	6	6	0
15.10.7.1 (source)	1	1	0
15.10.7.2 (global)	4	4	0
15.10.7.3 (ignoreCase)	4	4	0
15.10.7.4 (multiline)	4	4	0
RegExp grammar			
15.10.2.3 (disjunction)	17	17	0
15.10.2.5 (term)	6	6	0
15.10.2.6 (assertion)	44	44	0
15.10.2.7 (quantifier)	69	69	0
15.10.2.8 (atom)	60	60	0
15.10.2.9 (atom-escape)	4	4	0
15.10.2.10 (character-escape)	13	12	1
15.10.2.11 (decimal-escape)	7	5	2
15.10.2.12 (char-class-escape)	2	2	0
15.10.2.13 (char-class)	28	28	0
15.10.2.15 (char-class-ranges)	42	42	0
RegExp prototype			
15.10.6 (prototype-obj)	5	5	0
15.10.6 (prototype)	8	8	0
15.10.6.2 (exec)	61	61	0
15.10.6.3 (test)	38	38	0
15.10.6.4 (toString)	6	6	0
Total	519	516	3

Table B.4: Test262 test results for the ES5 RegExp object.