ECMA-SL - A Platform for Specifying and Running the ECMAScript Standard

Luís Miguel Alves Loureiro
luismaloureiro@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Portugal

Abstract

ECMAScript, commonly known as JavaScript, is one of the most widespread dynamic languages and it is the *de facto* language for client-side web applications. Due to its complexity, ECMAScript is both a hard language to understand by typical developers and a difficult target for static analyses. We present ECMARef, a reference interpreter for ECMAScript that follows the ECMAScript standard version 5.1 line-by-line and is thoroughly tested against Test262, the official ES5 conformance test suite. To this end, we introduce ECMA-SL: a dedicated intermediate language for ECMAScript analysis and specification. We also present ECMA-SL2English, a tool to generate the HTML English description of the standard from ECMARef. The resulting document is compared against the official document using classical text-based comparison metrics and HTML-specific metrics, obtaining high similarity scores using both classes of comparison metrics. On the whole, we believe that this project is a steppingstone towards the goal of automating the generation of the textual description of the standard.

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

1 Introduction

`ECMAScript`, commonly known as JavaScript, is one of the most widespread dynamic languages: it is the *de facto* language for client-side web applications; it is used for server-side scripting and it even runs on small embedded devices. It is used by 97.4% of websites[1], and is the most active language on GitHub[2] and the second most active on Stack-Overflow[3]. `ECMAScript` is specified in the `ECMAScript` standard [2], a long highly complex document written in English. Due to its complexity, `ECMAScript` is both a hard language to understand by typical developers and a difficult target for static analyses. For this reason, most program analyses for `ECMAScript` aim at limited, ad-hoc fragments of the language. Also, it is a constantly evolving language whose specification has been mostly growing every year.

The `ECMAScript` English standard is written as if it was the pseudo-code of an `ECMAScript` interpreter. The semantics of all commands is described in operational style, detailing each step of the evaluation. Hence, maintaining and extending the `ECMAScript` standard is a complex error-prone task that involves manually editing complex HTML documents with the textual description of the semantics of the language. For instance, when adding a new feature to the standard, one has to guarantee that this feature is compatible with the internal invariants maintained by the semantics of the language and, most importantly, that it does not break the behavior of previous features. Given the current size and complexity of the standard, such guarantees are extremely hard to get. Therefore, the `ECMAScript` committee has established a multi-step procedure for new feature proposals, which involves the creation of test suites and the implementation of multiple prototypes in `ECMAScript` engines, parsers, transpilers, type checkers, among others.

As the `ECMAScript` standard becomes more complex, it also becomes more difficult to manage and extend. Hence, we believe that the `ECMAScript` specification should be generated from a reference implementation of the language. This methodology would bring several benefits to the official specification of the language when compared to the current text-based methodology adopted by the `ECMAScript` committee; namely:

1. Writing code is easier than writing HTML pseudo-code following the conventions of the standard.
2. Making sure that a new change to the standard is backward compatible with previous versions is easier to achieve; for instance, one can run the extended reference interpreter on the official test suite and check if the new change causes tests to fail.
3. Generating test cases for newly introduced features can be done by applying automatic test generation techniques [9] to the reference interpreter focusing on the new features.
4. Measuring the coverage of the official test suite can be done simply by running the reference interpreter on it.

In this thesis, we demonstrate that it is possible to generate an HTML version of the `ECMAScript` standard from a

[1]Usage statistics of JavaScript as client-side programming language on websites, July 2021, W3Techs.com - https://w3techs.com/technologies/details/cp-javascript
[2]Github most active programming languages based on pull requests - https://madnight.github.io/githut/
[3]Stack Overflow Trends over time based on use of their tags - https://insights.stackoverflow.com/trends

reference implementation without significant changes to its text. To achieve this goal, we have implemented ECMARef5, a novel reference interpreter for ECMAScript that follows the English standard line-by-line, together with a tool that generates a faithful HTML version of the standard from the code of ECMARef5. Indeed, we believe that most ECMAScript developers would not be able to identify the official standard when presented with both versions of the standard (the official one and the one generated by our tool). Furthermore, the automatically generated version is superior to the official one in that it is more consistent in the use of language, with the same behaviours always described in the same way in similar contexts.

At the core of ECMARef5 is ECMA-SL, a dedicated intermediate language for ECMAScript analysis and specification. ECMA-SL is a simple language that supports all of the meta-constructs used in the standard to describe the semantics of ECMAScript programs. Hence, using ECMA-SL, we were able to implement ECMARef5 without departing from the pseudo-code of the standard. However, some of the programming language constructs included in ECMA-SL can be expressed using more fundamental constructs. Therefore, we have additionally designed a simpler intermediate language called Core ECMA-SL that we use as a compilation target for ECMA-SL.

2 ECMAScript standard

The ECMAScript Standard edition 5.1 [2] is the official document that defines the ECMAScript scripting language for version 5.1, hereafter referred to as ES5. The standard defines the types, values, objects, properties, functions, and program syntax and semantics that should exist in an ES5 language implementation. Note that the standard allows an implementation of the language to provide additional types, values, objects, properties, and functions.

2.1 Language Overview

We define the ES5 language in three main components: syntax and semantics; internal functions; and built-in objects. The syntax and semantics component groups all the syntactic grammars and semantics of expressions , statements , built-in types and also some lexical conventions, for instance, the definition of what is a white space, how comments are constructed and all the reserved words and keywords of the language.

The internal functions component include all functions that help define the semantics of the language. These are functions that are not exposed outside the scope of the language internals, that is, an ES5 program does not take advantage of any of these functions.

The final component is the one that contains all the built-in objects available whenever an ES5 program executes. These built-in objects include, for instance, the *Global object*, the *Function object*, and the *Array object*.

3 Related Work

The research literature covers a wide range of program analysis and instrumentation techniques for ECMAScript, such as: type systems [13, 21], abstract interpreters [14], points-to analyses [20], program logics [15, 16], operational semantics [10, 22, 24], intermediate representations/compilers [15], among others. Here we focus on operational semantics/reference interpreters for ECMAScript and intermediate languages and compilers for ECMAScript analyses.

There have been numerous research projects with the goal of formalising the semantics of ECMAScript including its built-in libraries. In the following, we review the most relevant of these projects.

JSCert. Bodin et al. [10] developed *JSCert*, the first mechanised specification of the ECMAScript semantics. The authors formalised a pretty-big-step semantics [11] of ES5 in the *Coq* [1] interactive proof assistant. Besides JSCert, the paper also describes *JSRef*, an ECMAScript reference interpreter defined in Coq and extracted from Coq to OCaml in order to be executed. The authors proved that JSRef is correct with respect to the formalised operational semantics and tested it against a fragment of *Test262* [6]. Later, Gardner et al. [17] extended the JSRef reference interpreter with support for ES5 Arrays. To this end, the authors linked JSRef to the *Google's V8* [8] Array library implementation. In this second paper, the authors additionally assessed the previous and current states of the JSCert and JSRef projects, providing a thorough analysis of the methodology as a whole and a detailed breakdown of the passing/failing tests. The JSCert project is especially relevant to us because it was the first project to emphasise the importance of having the code of a reference implementation matching the code of the corresponding standard. The authors proposed the concept of eye-ball closeness. We improve on this concept by removing the human out of the process, in that we can quantify the similarity between ECMARef5 and the official standard using the ECMA-SL2English HTML generator.

JSExplain. Charguéraud et al. presented *JSExplain* [12], a reference interpreter for the ES5 language that closely follows the text of the specification. JSExplain was written in a custom-made purely functional language with a built-in monadic operator for automatically threading the implicit state of the interpreter across pure computations. The authors further implemented a translator from their functional language to ECMAScript, allowing them to run JSExplain in the browser. The main goal of JSExplain is to allow programmers to debug the execution of ECMAScript programs, having access to both the state of the program and the internal state of the ECMAScript interpreter. In other words, with JSExplain, we can not only code-step the execution of an ECMAScript program but also the execution of the ECMAScript interpreter itself. While the goals of JSExplain are very close to our own, important differences remain: (1) JSExplain was not tested against Test262, and (2) the authors do not quantify the similarity between their reference interpreter and the official text of the standard.

4 ECMA-SL

Our main goal with the design of ECMA-SL was to obtain the simplest possible intermediate language that would allow us to implement the ECMAScript standard in a faithful way. To this end, we included in ECMA-SL all the meta-constructs of the ECMAScript standard in order to implement an ECMAScript reference interpreter that matches the pseudo-code of the standard line-by-line. However, some of these meta-constructs can be expressed using more fundamental constructs. For instance, the standard makes use of a repeat statement and a foreach statement which can both be modelled using a simple while statement. Hence, we have designed a simpler intermediate language called Core ECMA-SL that we use as a compilation target for ECMA-SL. On the whole, our ECMA-SL engine comes with: (1) an ECMA-SL parser, (2) a compiler from ECMA-SL to Core ECMA-SL, and (3) a Core ECMA-SL interpreter. All three modules were written in the OCaml programming language [5].

4.1 Designing the ECMA-SL Language

ECMA-SL is a simple imperative language that retains the fundamental dynamic behavior of ECMAScript: (1) dynamic function calls, (2) dynamic creation and deletion of object properties, and (3) dynamic code evaluation. An ECMA-SL program is simply a set of top-level functions with a designated entry-point function called main. All ECMA-SL expressions are standard with the exception of static and dynamic property lookup expressions, function calls with a catch clause, and external function calls, which we explain below:

- *Property lookup expressions*: In ECMA-SL, there are two types of property lookup expressions: static and dynamic. For static property lookup expressions the name of the property being inspected is known at static time (e.g. o.foo). In contrast, for dynamic property lookup expressions, it must be dynamically computed (e.g. o[y]).
- *Function calls with a catch clause*: In ECMA-SL, function calls may be extended with a catch clause that specifies an error handler h to process the outcome of the corresponding function in case it throws an error. In such cases, the whole function call expression evaluates to the return of the error handler h. For instance, consider the call f(3) catch h. If the body of function f throws an error, the ECMA-SL engine executes the handler h giving it as input the error thrown by f, with the whole expression evaluating to the return of h.
- *External function calls*: Finally, external function calls provide a mechanism for seamlessly extending the semantics of the ECMA-SL language without having to change its syntax. More concretely, external function calls allow for the execution of functions directly implemented in OCaml. These functions can, in turn, call arbitrary system commands, executing programs written in other languages. As an example, we have used an external function called parseJS in our implementation of the ECMAScript standard. This function is used to dynamically parse ECMAScript programs in text format. When interpreting the call extern parseJS(str), the ECMA-SL engine first checks if it contains an external function called parseJS. If it does, it then executes that function on the string argument given as input.

4.2 Compiling ECMA-SL to Core ECMA-SL

As ECMA-SL contains several programming-language constructs that can be expressed using more fundamental constructs, we created a simpler version of ECMA-SL, called Core ECMA-SL, together with a compiler from ECMA-SL to Core ECMA-SL. With this compiler, instead of interpreting an ECMA-SL program directly, one first compiles it to Core ECMA-SL and then interprets the obtained program.

Core ECMA-SL differs from ECMA-SL in the following aspects:

1. It only allows for side-effect-free expressions that do not interact with the heap, i.e. Core ECMA-SL expressions can only interact with the variable store;
2. It contains a single loop statement (no repeat, repeat-until, and foreach statements);
3. It contains a single conditional statement (no switch and match statements);
4. It does not support global variables;
5. It does not include an error-handling mechanism (no throw statement and no function call with catch clause).

Since Core ECMA-SL expressions do not have side-effects and cannot interact with the object heap, the ECMA-SL expressions with these features were "promoted" to equivalent Core ECMA-SL statements. For instance, while ECMA-SL contains a property lookup expression of the form ⟨expr⟩ ‘[’ ⟨expr⟩ ‘]’, Core ECMA-SL contains a single dedicated property lookup assignment of the form ⟨var⟩ := ⟨expr⟩ ‘[’ ⟨expr⟩ ‘]’.

The ECMA-SL to Core ECMA-SL compiler is structured in a modular fashion with compilation functions for ECMA-SL programs (compile_prog), functions (compile_func), statements (compile_stmt), and expressions (compile_expr). The function compile_prog creates a new Core ECMA-SL program with the compiled functions of the original ECMA-SL program. Analogously, the function compile_func creates a new Core ECMA-SL function with the compiled statements of the original ECMA-SL function. The functions compile_stmt and compile_expr are more involved as they have to model the behavior of ECMA-SL statements and expressions using the simpler statements and expressions of Core ECMA-SL.

5 Implementing ECMAScript in ECMA-SL

In this chapter, we explain the internal representations used in ECMARef5 to model the different types of artifacts used in the ECMAScript standard (5.1), followed by the description of the implementation of ECMAScript built-in objects and initial heap (5.2). Finally, we demonstrate how ECMARef5 follows the ECMAScript standard line-by-line (5.3) and provide an overview of the compilation of ECMAScript programs to ECMA-SL (5.4).

5.1 ECMARef5 Internal Representations

We start by briefly explaining the internal representations that we have used to model different types of artifacts used in the ECMAScript standard. More specifically, we discuss our internal representations of: (1) ECMAScript objects and (2) property descriptors.

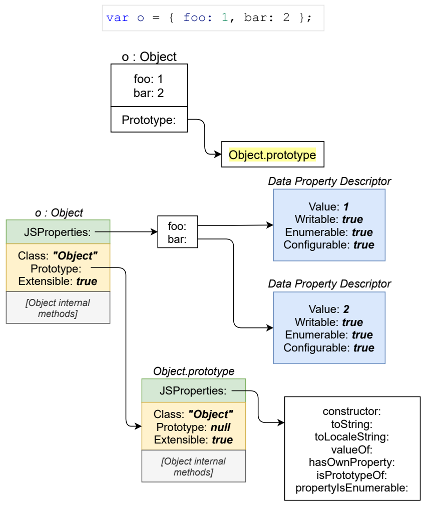


Figure 1. ECMAScript (top) and ECMA-SL (bottom) objects representations.

ECMAScript objects. ECMAScript objects can be thought of as key-value dictionaries mapping properties to values. Each ECMAScript object has a set of internal properties, providing meta-information about the object, and a set of named properties, which are explicitly controlled by the programmer. In ECMA-SL, we represent every ECMAScript object as two distinct objects: one main object storing the internal properties of the original ECMAScript object, and one auxiliary object storing its named properties. The main object has a dedicated property, JSProperties, which points to the auxiliary object. For instance, Figure 1 shows an ECMAScript object on the left and its representation in ECMA-SL on the right. One can see that object o is split into two objects: the one that keeps its internal properties and the one that keeps its named properties.

Importantly, we have to associate two objects with each ECMAScript object to avoid clashes between named properties and internal properties. Suppose that, instead, we used a single object containing all the named properties and internal properties of a given ECMAScript object. Furthermore, suppose that one of the named properties of the original object was, for instance, the property Class. In this situation, how could we distinguish the internal property Class from the named property Class? We automatically avoid this type of clash by keeping the named properties in a separate object.

Property descriptors. ECMAScript objects contain two types of properties: internal properties and named properties. The named properties are stored in the auxiliary object which is accessed through the property JSProperties. The ECMAScript standard mandates that named properties be represented by *Property Descriptors*. Depending on the attributes contained in the record, property descriptors are classified as data property descriptors or accessor property descriptors.

In ECMA-SL, we represent property descriptors as objects which store the attributes defined in the ECMAScript standard: [[Value]] and [[Writable]] for data property descriptors; [[Get]] and [[Set]] for accessor property descriptors; and, [[Enumerable]] and [[Configurable]] for both types of property descriptor. We explain our internal representation of property descriptors by appealing to the example given in Figure 1. Here, we represent property descriptors as blue boxes containing their corresponding attributes. Specifically, we have two property descriptors, respectively storing the values of the properties foo and bar of object o. Each descriptor stores the value of the corresponding ECMAScript named property in its property Value. Additionally, each descriptor has the three meta-properties that fully populate the corresponding data property descriptor: Writable, Enumerable, and Configurable.

5.2 ECMARef5 Built-ins and Initial Heap

The ECMAScript standard defines a comprehensive runtime library, which provides a large number of utility functions to operate on objects, functions, primitive types, and regular expressions. These functions are made available to the programmer via dedicated built-in objects and they are created in the heap whenever an ECMAScript program executes.

Built-in objects. Our ECMAScript interpreter fully supports all the ECMAScript built-in objects except for the Global Object, which is still not yet fully implemented. However, not all built-in objects were implemented in the context of this thesis.

Initial heap. The standard does not specify how built-in objects are created in the initial heap; instead, it simply states that they must be there. In contrast, `ECMARef5` creates the built-in objects in the heap by calling the function `initGlobalObject`. The creation of the initial heap is not straightforward because of the mutual dependencies between built-in objects. In order to cope with these mutual dependencies, we have to postpone the creation of certain properties when initialising their corresponding objects. Instead of creating the initial heap programmatically, constructing one built-in object at a time and carefully establishing the dependencies between them, one can instead load the initial heap to memory from a pre-computed serialised version.

5.3 Line-by-line Closeness

We demonstrate that `ECMARef5` follows the `ECMAScript` standard line-by-line by example. We consider the internal function `[[GetProperty]]` presented in Figure 2.

Implementing the internal functions of the `ECMAScript` standard in `ECMA-SL` is straightforward given that `ECMA-SL` contains syntactic constructs corresponding to all the meta-constructs used in the standard. However, in order to streamline the interaction with our internal representation of ES5 objects, we make use of a range of auxiliary functions, such as `getInternalProperty(O, P)` for obtaining the internal property `P` of `O`.

It is worth noting the use of dynamic function calls in our `ECMA-SL` implementation of the `[[GetProperty]]` internal function. The expression `{O.GetOwnProperty}(O, P)` will call the function bound to the property `GetOwnProperty` of `O` with parameters `O` and `P`.

5.4 Compiling ECMAScript to ECMA-SL

The compilation of an `ECMAScript` program to `ECMA-SL` and its interpretation are organised as an execution pipeline that comprises the following three steps:

1. compiling the input program to ECMA-SL, storing the resulting code in a file called `out.esl`;
2. compiling the file `out.esl` to Core ECMA-SL obtaining the file `core.cesl`;
3. interpreting the obtained `Core ECMA-SL` program using our `ECMA-SL` interpreter.

`ECMA-SL` comes with two execution pipelines: a non-optimised one and an optimised one. The main difference between these two pipelines pertains to the loading of the `ECMAScript` initial heap. While the non-optimised pipeline builds the initial heap via the execution of the function `initGlobalObject`, described in Subsection 5.2, the optimised pipeline loads it directly to memory from a pre-generated JSON file with its contents. Below we describe the three main phases of our execution pipelines and detail the design of the optimised pipeline.

JS2ECMA-SL. Given a file containing an `ECMAScript` program, we first pass it to the `JS2ECMA-SL` compiler. `JS2ECMA-SL` parses the given program using *Esprima* [3]. The AST of the given program generated by Esprima is then transformed into an `ECMA-SL` program that recreates it in `ECMA-SL`. For instance, the `ECMAScript` program `x = 2` is transformed into the `ECMA-SL` function `buildAST`. This function simply creates the AST of the given program in the `ECMA-SL` heap. In order to obtain an `ECMA-SL` program that actually emulates the behaviour of the original `ECMAScript` program, we must call the `ECMAScript` interpreter on the result of `buildAST`. To this end, we generate the program `out.esl`, which imports both the ES5 interpreter and the generated `buildAST` function calling the interpreter on the result of `buildAST`.

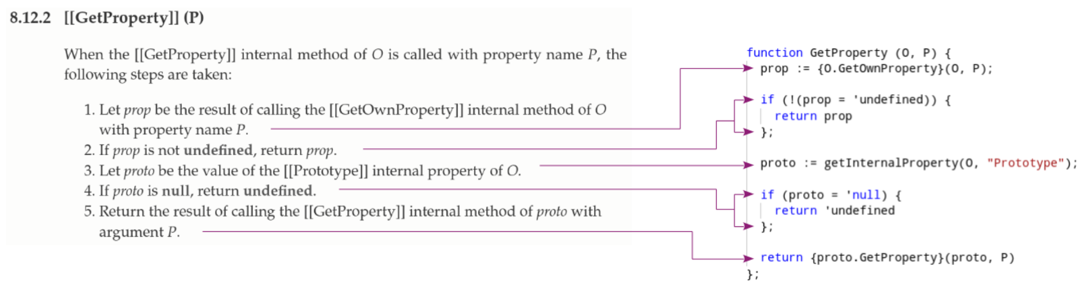
ECMA-SL2Core. The obtained `ECMA-SL` program is compiled to `Core ECMA-SL` using the compiler introduced in Chapter 4 resulting in the file `core.cesl`. All the imports included in the file `out.esl` are resolved as part of the compilation to `Core ECMA-SL`. Hence, the returned program is completely self-contained, including all the code of the ECMAScript interpreter as well as the code of the `buildAST` function of the program to be run.

ECMA-SL Interpreter. The obtained `Core ECMA-SL` program is interpreted using our `ECMA-SL` Interpreter written in OCaml. The interpreter has two main execution modes: silent and verbose. In silent mode, the interpreter outputs the final `ECMA-SL` heap generated by executing the program. In verbose mode, the interpreter additionally logs the sequence of executed commands for debugging purposes.

Optimised Pipeline. When interpreting an `ECMAScript` program, one starts by constructing the initial `ECMAScript` heap, which contains all the `ECMAScript` built-in objects. The simplest way to set up this initial heap is to actually execute the `ECMA-SL` code that constructs its objects. This involves the execution of thousands of `ECMA-SL` commands, often taking a significant amount of time when compared to the amount of time taken by the execution of the whole program. However, the initial heap is always the same. This means that we do not need to recompute it every time we execute a compiled `ECMAScript` program. To this end, we designed an optimised version of the execution pipeline that, instead, loads a previously generated and JSON serialised version of the initial heap. This additional step is performed before start the interpretation of the `Core ECMA-SL` program.

6 HTML Generator

Using `ECMA-SL2English`, we demonstrate that is possible to generate the `ECMAScript` standard from a reference implementation without significant changes to its text. Indeed, we believe that most `ECMAScript` developers would not be able to tell the difference between the version of the standard generated by our tool and the original one. Furthermore,



8.12.2 [[GetProperty]] (P)

When the [[GetProperty]] internal method of *O* is called with property name *P*, the following steps are taken:

1. Let *prop* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. If *prop* is not **undefined**, return *prop*.
3. Let *proto* be the value of the [[Prototype]] internal property of *O*.
4. If *proto* is **null**, return **undefined**.
5. Return the result of calling the [[GetProperty]] internal method of *proto* with argument *P*.

```
function GetProperty (O, P) {
  prop := {O.GetOwnProperty}(O, P);

  if (!(prop = 'undefined)) {
    return prop
  };

  proto := getInternalProperty(O, "Prototype");

  if (proto = 'null) {
    return 'undefined
  };

  return {proto.GetProperty}(proto, P)
};
```

Figure 2. The specification of the Object internal function `[[GetProperty]]` and the corresponding ECMA-SL code.

the automatically generated version is superior to the original one in that it is more consistent in the use of language; the same behaviour is described in the same way in similar contexts. This is not the case of the actual standard where, even in analogous contexts, the same behaviour can be described in different ways. For instance, consider the following four different descriptions of a call to the internal method `[[GetOwnProperty]]`, where we underline the differences between the four:

1. *Let ownDesc be the result of calling the [[GetOwnProperty]] internal method of O with argument P.*
2. *Let prop be the result of calling the [[GetOwnProperty]] internal method of O with property name P.*
3. *Let desc be the result of calling the [[GetOwnProperty]] internal method of O with P.*
4. *Let desc be the result of calling the [[GetOwnProperty]] internal method of O passing P as the argument.*

There is no special context in which any one of these function calls occur, so there is no need to have different descriptions for the same behaviour.

Although `ECMA-SL` is very close to the language of the pseudo-code of the standard, the design of `ECMA-SL2English` was not straightforward. We highlight two main challenges:

- The use of phrases that cannot be inferred from the code of the interpreter to describe specific behaviours of `ECMAScript`; for instance, step 4 of the pseudo-code of the `[[Put]]` internal method appears as follows: *"Let desc be the result of calling the [[GetProperty]] internal method of O with argument P. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor"*. The second sentence is merely informative as it describes the expected result returned from calling `[[GetProperty]]`; hence, it cannot be inferred by our reference implementation.
- The use of different syntactic constructions/HTML structures to describe the same behaviour in different contexts; for instance, if-else statements have different HTML representations throughout the standard.

6.1 Code Generation Algorithm

The HTML generation of the `ECMAScript` standard from a reference interpreter written in `ECMA-SL` involves the execution of several preliminary steps, which guarantee that all the `ECMA-SL` code that is to be transformed is correctly organised and does not contain language constructs that are unrecognised by the `ECMA-SL2English` tool.

The main algorithm of `ECMA-SL2English`, which is composed of the following steps:

1. Filter out all the `ECMA-SL` code that is not to be transformed into HTML;
2. Normalise the code to be generated so as to facilitate the code generation process;
3. Sort all the interpreter functions by section/subsection identifier;
4. Load all the HTML rules created in JSON format;
5. Transform the `ECMA-SL` code into HTML.

6.2 Directives and Rules

In order to allow for greater flexibility during the HTML generation process, we extend `ECMA-SL` with a set of code generation directives and make our code generation algorithm parametric on a set of implementation-independent code generation rules to be fed to the tool in JSON format.

6.2.1 Code Annotations. We extended the `ECMA-SL` syntax with five main code generation directives:

1. **Function signature directive**: The function signature directive is used to provide additional metadata about the function that it annotates. More concretely, it includes: (1) the corresponding `ECMAScript` standard section number; (2) the HTML text with the description of the corresponding algorithm; (3) the HTML text containing further notes and details about the algorithm; and (4) the title of the section of the standard where the function is contained.
2. **Statement wrapper directive**: This directive is used to add more text to the HTML code generated for the enclosed statement. The extra text can be either prepended or appended to the generated HTML code. We use the syntax `gen_wrapper [before:str] { s`

} to prepend the string `str` to the HTML code generated for statement `s`. Analogously, we use the syntax `gen_wrapper [after:str] { s }` to append the string `str` to the generated HTML code.
3. **Then and else directives**: The then and else directives are used to annotate `ECMA-SL` if-then-else statements. These annotations can be added to either the then clause or the else clause, extending these clauses with a code generation directive with the following format: `[keyword]:[HTML]`. In addition to the `before` and `after` keywords, already seen for the statement wrapper, we also have the keyword `replace-with`, which is used to replace the default HTML code with the provided HTML code.
4. **For-each directive**: With the for-each directive we follow the same approach as the one we used with the statement wrapper and the then and else directives, applying the same ideas to the `ECMA-SL` for-each statement. We annotate this statement with a code generation directive with the format `[keyword]:[HTML]`, where the possible keywords are `before` and `after`.
5. **Match pattern directive** The match pattern directive is used to annotate pattern clauses of the `ECMA-SL` match statements. The pattern clauses are mainly used to implement semantic productions of expressions and statements. Analogously to the function signature directive, the match pattern directive includes metadata for: (1) the corresponding `ECMAScript` standard section number; (2) the description of the semantic production; (3) the HTML text with further notes; and (4) the title of the section of the standard.

6.2.2 JSON Rules. While most `ECMA-SL` functions and operators used in `ECMARef5` can be turned into HTML in a straightforward way, some `ECMA-SL` functions and operators have specific textual descriptions and patterns associated with them. As we did not want to hard-code specific textual descriptions in our implementation of the HTML code generator, we decided to make the code generator parametric on a set of *implementation-independent* code generation rules to be fed to the tool in JSON format. The support for code generation rules makes `ECMA-SL2English` a highly flexible tool as it allows for the addition of new rules without modifying its code base.

`ECMA-SL2English` includes three types of code generation rules: function call rules, operator rules, and property lookup rules. These rules are specified in JSON format and loaded into the code generator before the starting of the code generation process. Code generation rules can be seen as string templates whose placeholders are going to be filled with strings computed at code generation time. For instance, the rule for `AbstractEqualityComparison` can be seen as the following string template:

```
the result of performing abstract
   equality comparison {0} == {1}
```

where `{0}` and `{1}` are placeholders to be replaced with the text generated for the first and the second arguments given to the function call, respectively.

7 Evaluation

In this chapter, we evaluate the main outcomes of this thesis: `ECMARef5`, our `ECMAScript` interpreter written in `ECMA-SL`, and `ECMA-SL2English`, our HTML generator tool with which we obtain an HTML version of the `ECMAScript` standard directly from the code of `ECMARef5`.

7.1 ECMARef5 Evaluation

`ECMARef5` was tested against `Test262` [6], the official `ECMAScript` test suite. `Test262` is comprised of thousands of test files, often including multiple test cases per test file.

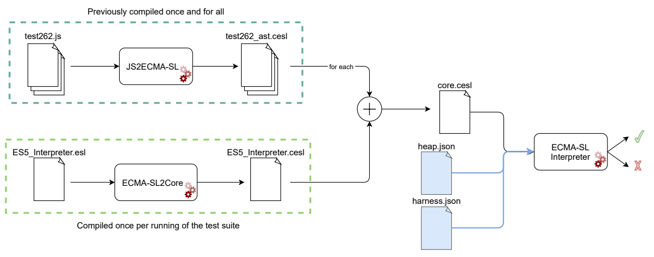


Figure 3. Test262 test fully optimised execution pipeline.

7.1.1 Testing pipeline. In order to streamline the testing process, we have developed an optimised version of the testing pipeline. Figure 3 illustrates the optimised testing pipeline, which is based on the following four main optimisations:

1. `Test262` tests are compiled directly to `Core ECMA-SL` once and for all. More specifically, we have compiled all `Test262` tests directly to `Core ECMA-SL` and stored their compilation for later use. We do not need to recompile tests because their code never changes.
2. `ECMARef5` is compiled to `Core ECMA-SL` only one time per running of the `Test262` test suite. In contrast to the code of the test files, which always stays the same, `ECMARef5` is an evolving project whose code base is frequently changed. Hence, every time we want to run the test suite, we must recompile `ECMARef5` to `Core ECMA-SL`. However, as the same interpreter runs for all tests, we only need to do it once per running of the test suite. Given a test file, the final `Core ECMA-SL` program to be executed is obtained by concatenating its compilation to `Core ECMA-SL` with the compiled code of `ECMARef5`.
3. The initial heap is loaded directly to memory from a pre-generated JSON serialisation. This initial heap contains all the built-in objects and their corresponding

function objects. However, much like the code of the test files, the initial heap is always the same. Hence, we do not need to re-compute it every time a test is run. Instead, we generate a JSON file with the serialisation of the ECMAScript initial heap in ECMA-SL once and for all and we load this file directly to memory at the start of the interpretation of an ECMAScript program.

4. The harness AST is loaded directly to memory from a pre-generated JSON serialisation. Much like the initial heap, the code of the harness never changes and is shared by all Test262 tests.

7.1.2 Testing results. Table 1 presents the breakdown of the testing results per section of the ECMAScript standard. The results show that ECMARef5 passes 12,026 tests out of 12,068 applicable tests. In total, only 42 tests are currently failing; of these, only 22 tests pertain to sections that were implemented in the context of this thesis. From these 22 failing tests, the two that pertain to Section 7 have two distinct reasons for failing: one fails because we still have issues regarding the parsing of unicode escape sequences, and the other one is impossible to solve, since the AST created by Esprima returns an error message that is handled by ECMARef5 as a *ReferenceError* and the test is annotated as throwing a *SyntaxError*. The tests that are failing in Section 15.7 indicate that ECMARef5 has implementation issues in some of the Number.prototype functions. Additionally, the implementation of one of these functions is causing one test to fail in Section 11. At the time of writing, we were not able to identify the reasons for the failing tests in Section 10 and Section 15.3.

Table 1 also presents the total execution time per section. For each section we show the times obtained using both the optimised and the non-optimised execution testing pipelines, clearly demonstrating that the implemented optimisations were instrumental to streamline the testing process. They allowed us to achieve an overall 295% performance boost and an average 309% performance boost. These times were obtained using a machine with an Intel Core i7-3610QM 2.3GHz, DDR3 RAM 12GB, and a 256GB solid-state hard-drive running Manjaro Linux.

Section	#Tests			Time	
	Total	Passed	Failed	Opt.	Non-Opt.
7	545	543	2	02m36s	08m10s
8	184	184	0	00m55s	03m00s
9	115	115	0	00m30s	01m50s
10	414	413	1	02m08s	06m49s
11	1635	1634	1	09m20s	27m56s
12	648	648	0	03m16s	10m12s
13	228	228	0	01m12s	03m42s
14	24	24	0	00m06s	00m23s
15.1	195	195	0	01m48s	04m05s
15.2	2885	2885	0	14m35s	47m44s
15.3	411	410	1	02m22s	07m02s
15.4	2268	2268	0	15m12s	41m21s
15.5	861	856	5	04m24s	14m14s
15.6	34	34	0	00m09s	00m32s
15.7	191	174	17	00m53s	03m04s
15.8	171	171	0	01m14s	03m10s
15.9	539	533	6	02m44s	08m55s
15.10	520	513	7	05m56s	11m52s
15.11	84	84	0	00m26s	01m23s
15.12	116	114	2	00m40s	02m00s
Sub-Total	7764	7742	22	41m30s	2h09m02s
Total	12068	12026	42	1h10m26s	3h27m44s

Table 1. Test262 testing results per section of the ECMAScript standard. Sub-total shows the results that apply in the context of this thesis.

7.2 ECMA-SL2English Evaluation

In this section we evaluate the closeness of the generated ECMAScript standard to the official HTML version of the standard. To this end, we make use of both classical text-based comparison metrics [18] as well as HTML-specific metrics based on the concept of tree similarity [26].

7.2.1 Text-based metrics. Text-based metrics are used to measure the similarity or dissimilarity between two given character strings. More concretely, given two character strings, text-based comparison algorithms compute a number that represents the (dis)similarity between the two given strings. Text-based metrics can be broadly divided into two main groups: *character-based metrics*, which take into account the order in which individual words (also referred to as tokens) appear within the two given character strings, and *token-based metrics*, which ignore that order. Here, we focus on character-based text-comparison metrics as the order in which words appear in the standard is relevant to us.

Most character-based text-comparison metrics are based on variations of the popular *Edit distance* algorithm [23].

7.2.2 HTML-specific metrics. HTML-specific metrics are calculated using two different measures: *structural similarity* and *style similarity*. The structural similarity applies a sequence comparison algorithm to the lists of HTML tags existing in both HTML documents. This algorithm is adapted from the *Gestalt Pattern Matching* [25] and the idea is to find the longest contiguous matching subsequence. The style similarity calculates the *jaccard similarity coefficient* [19] between the CSS classes existing in both HTML documents.

7.3 Scope and Granularity

The HTML generator was only applied to parts of ECMARef5 developed in the context of this thesis. Furthermore, we excluded Sections 15.6, 15.7, 15.8, and 15.11. The reason for not applying the ECMA-SL2English to the entire ECMARef5 implementation is that in order to obtain good results one must annotate the implementation with code generation directives and provide the appropriate code generation rules. This is a time-consuming task that we could not carry out in the time-frame of this thesis and therefore leave for future

Section	L	JW	S95	NW	G	SW
8	88.2	91.1	91.2	91.6	**94.4**	84.1
9	74.3	87.9	88.0	76.9	**88.8**	66.0
10	82.3	90.2	90.2	85.7	**92.5**	**92.5**
11	86.1	90.9	90.9	88.1	**94.6**	84.2
12	82.2	90.0	90.0	85.1	**92.8**	78.1
13	81.6	89.4	89.5	84.4	**92.9**	76.2
14	64.0	86.8	**86.9**	69.2	85.7	65.5
15.1	81.9	89.7	89.7	84.1	**93.2**	75.9
15.2	86.7	90.8	90.8	89.1	**94.5**	82.6
15.3	81.0	89.8	90.0	84.8	74.9	**91.9**
Average	80.8	89.7	89.7	83.9	90.4	79.7

Table 2. Results of the application of some Edit Distance algorithms to sections of the `ECMAScript` standard generated by the `ECMA-SL2English` tool. The acronyms `L`, `JW`, `S95`, `NW`, `G`, and `SW` respectively mean Levenshtein, Jaro-Winkler, Strcmp95, Needleman-Wunsch, Gotoh, and Smith-Waterman

Section	Scores			#Lines	
	Style	Structural	Joint	Official	Generated
8	94.9	90.2	92.5	3731	3737
9	75.6	88.4	82.0	1022	1137
10	96.6	44.0	70.3	3961	4123
11	92.6	80.7	86.6	8999	8862
12	97.4	70.6	84.0	4714	4392
13	96.6	90.3	93.2	1404	1353
14	100	82.1	91.1	298	253
15.1	91.6	90.5	91.1	307	330
15.2	97.6	93.3	95.4	2232	2270
15.3	100	86.6	93.3	1627	1746
Average	94.3	81.7	88	-	-
Total	-	-	-	28295	28203

Table 3. Results of the application of HTML similarity to sections of the `ECMAScript` standard generated by the `ECMA-SL2English` tool.

work. Here we focus on the core sections and built-in objects of the standard.

7.4 Evaluation Pipeline

It is not practical to apply the selected text-comparison algorithms to entire sections of the standard since most of these algorithms have quadratic asymptotic complexity and therefore would exhibit prohibitive execution times. Hence, to obtain the evaluation results for each section of the standard, we first have to pre-process the two HTML documents, the generated one and the official one. More concretely, we split both documents into two sets of files, with each file containing a single algorithm/function of the standard, and apply all text-comparison algorithms at the standard-algorithm granularity level; then, we combine the obtained results using a weighted arithmetic average. The complete evaluation pipeline is divided into the following three components:

1. The **Splitter** component gets both HTML documents and splits them into multiple HTML files, with each file containing a single algorithm of the standard;
2. The **Calculator** component applies all the HTML similarity and Edit distance algorithms to each pair of HTML files created by the Splitter component. The Calculator returns a list with all the computed results together with the number of lines and characters of the given standard algorithm;
3. The **Aggregator** component calculates the final results for each section of the standard by computing the arithmetic average of the results generated in the previous step.

7.5 Text-based Metrics

In order to compute the similarity between the official and the generated versions of the standard, we make use of the `textdistance` open-source project [7]. This project comes with nine different variations of the edit distance algorithm, of which we use the following six: (1) *Levenshtein*, (2) *Jaro-Winkler*, (3) *Needleman-Wunsch*, (4) *Smith-Waterman*, (5) *Gotoh*, and (6) *strcmp95*.

Results. We present the overall results for the six selected metrics in Table 2. For each section of the standard, we highlight in bold the highest value and underline the lowest one. Excluding the scores obtained using the Smith-Waterman algorithm and with the exception of Sections 9 and 14, the obtained results are consistently high for all metrics (always above 80%). It is important to note that Sections 9 and 14 represent a small fragment of the total amount of generated text (≈5%).

7.6 HTML-specific Metrics

We have used the HTMLSimilarity open-source project [4] to compute the structural similarity and the style similarity between the official and the generated versions of the standard. Results are presented in Table 3. The measured structural and style similarities are generally high except for the cases of Sections 10 and 12. These two sections have their respective structural similarities highly affected by lower values computed for some of their enclosed subsections. The common characteristic of these enclosed subsections (10.5, 10.6, and 12.11) is that they are substantially larger than a typical subsection of the standard. We observed that differences in structure tend to have a non-linear impact on the computed structural similarity score. Hence, the larger a subsection is, the less likely it is to have a good similarity score. We could have fixed this issue by splitting these subsections into smaller fragments, computing the similarity score for each fragment, and combining the obtained scores using a weighted average. We chose not to do this to keep our evaluation methodology consistent across all the subsections of the standard.

8 Conclusions

In this thesis, we have demonstrated that it is possible to generate the ECMAScript standard from a reference implementation without significant changes to its text. The integration of such a tool into the ECMAScript standardisation pipeline would bring several benefits; in particular, it would simplify both the specification process and the testing of new features.

We developed this project as part of a wider project whose goal is to build a tool-suite for ECMAScript analysis and specification based on ECMA-SL. We contributed to the overarching ECMA-SL project in three different ways. First, we designed the ECMA-SL and Core ECMA-SL languages and developed the compiler from ECMA-SL to Core ECMA-SL. Second, we developed ECMARef5, a new ES5 reference interpreter that follows the standard line-by-line; importantly, ECMARef5 is, to the best of our knowledge, the most complete academic reference implementation of the fifth version of the ECMAScript standard. Third, we designed ECMA-SL2English, our HTML code generator that creates an HTML version of the standard from the code of ECMARef5. With ECMA-SL2English we were able to measure the closeness between ECMARef5 and the official ES5 standard using out-of-the-box text-based comparison metrics.

The two main outcomes of this thesis, ECMA-SL2English and ECMARef5, were thoroughly evaluated. ECMARef5 was tested against *Test262* [6] passing 12,026 tests out of 12,068 applicable tests. Importantly, ECMARef5 is the most complete reference implementation of the ES5 standard, with JS-2-JSIL [15], the second most complete, only passing 8,797 tests. We evaluated ECMA-SL2English by comparing the generated standard against the official one using both text-based and HTML-based comparison metrics. We have obtained consistently high scores for both metrics (always above 80% similarity score).

References

[1] [n.d.]. *Coq - Interactive formal proof management system.* https://coq.inria.fr/ Accessed on 2021-09-23.

[2] [n.d.]. *ECMAScript® Language Specification, 5.1 Edition / June 2011.* http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf Accessed on 2021-09-23.

[3] [n.d.]. *Esprima - a high performance, standard-compliant ECMAScript parser written in ECMAScript.* https://esprima.org/ Accessed on 2021-09-23.

[4] [n.d.]. *HTMLSimilarity - Compare html similarity using structural and style metrics.* https://github.com/matiskay/html-similarity Accessed on 2021-07-27.

[5] [n.d.]. *OCaml - General-purpose, multi-paradigm programming language.* https://ocaml.org/ Accessed on 2021-09-23.

[6] [n.d.]. *Test262 - Official ECMAScript Conformance Test Suite.* https://github.com/tc39/test262/ Accessed on 2021-09-23.

[7] [n.d.]. *TextDistance - python library for comparing distance between two or more sequences by many algorithms.* https://github.com/life4/textdistance Accessed on 2021-07-13.

[8] [n.d.]. *V8 - Google's open source high-performance JavaScript and WebAssembly engine, written in C++.* https://v8.dev/ Accessed on 2021-09-23.

[9] Saswat Anand, E. Burke, T. Chen, John A. Clark, Myra B. Cohen, W. Grieskamp, M. J. Harrold, A. Bertolino, Juan Li, and H. Zhu. 2013. An Orchestrated Survey on Automated Software Test Case Generation I.

[10] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 49, 87–100. https://doi.org/10.1145/2578855.2535876

[11] Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

[12] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JS-Explain: A Double Debugger for JavaScript. *WWW '18: Companion Proceedings of the The Web Conference 2018*, 691–699. https://doi.org/10.1145/3184558.3185969

[13] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (08 2017), 48:1–48:30. https://doi.org/10.1145/3133872

[14] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. IEEE Computer Society, 34–45. https://doi.org/10.1109/CGO.2015.7054185

[15] Jose Fragoso Santos, Petar Maksimović, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages* 2 (12 2017), 1–33. https://doi.org/10.1145/3158138

[16] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. *Sigplan Notices - SIGPLAN* 47, 31–44. https://doi.org/10.1145/2103621.2103663

[17] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. 2015. A Trusted Mechanised Specification of JavaScript: One Year On, Vol. 9206. 3–10. https://doi.org/10.1007/978-3-319-21690-4_1

[18] Aly H. Gomaa, Wael ; A. Fahmy. 2013. A Survey of Text Similarity Approaches. *International Journal of Computer Applications* 68. Issue 13. https://doi.org/10.5120/11638-7118

[19] P. Jaccard. 1901. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles* (1901), 241–272.

[20] Dongseok Jang and Kwang-Moo Choe. 2009. Points-to analysis for JavaScript. *Proceedings of the ACM Symposium on Applied Computing*, 1930–1937. https://doi.org/10.1145/1529282.1529711

[21] Simon Jensen, Anders Møller, and Peter Thiemann. 2009. *Type Analysis for JavaScript.*

[22] Sergio Maffeis, John Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. 307–325. https://doi.org/10.1007/978-3-540-89330-1_22

[23] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.* 33, 1 (March 2001), 31–88. https://doi.org/10.1145/375360.375365

[24] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. *ACM SIGPLAN Notices* 50 (06 2015), 346–356. https://doi.org/10.1145/2813885.2737991

[25] John W. Ratcliff and David E. Metzener. 1988. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal* (July 1988), 46.

[26] Hangjun Xu. 2014. *An Algorithm for Comparing Similarity Between Two Trees.* Master's thesis. Duke University.