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

**Luís Miguel Alves Loureiro**

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 Luís Brinquete Borbinha
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Members of the Committee: Prof. João Costa Seco

**September 2021**

# Acknowledgments

I would like to thank my supervisor, Prof. José Fragoso Santos, for his encouragement and confidence in the direction of the project. His continued support and tremendous availability were of great importance for the success of this thesis.

Also, I would like to thank my friends and all those that have supported me during the time of the development of the dissertation and that, inclusively, have helped me with the revision of some of my texts. Their encouragement helped me push forward in my darkest times.

**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 ECMARef5, 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 ECMARef5. 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

**Resumo**

ECMAScript, vulgarmente conhecida como JavaScript, é uma das linguagens dinâmicas mais difundidas e é a linguagem *de facto* para aplicações web que se executam num browser. Devido à sua complexidade, ECMAScript é uma linguagem difícil de compreender pelos programadores e um alvo difícil para análises estáticas. Apresentamos ECMARef5, um interpretador de referência para ECMAScript que segue a versão 5.1 da linguagem linha-a-linha e que é testado com bastante pormenor usando Test262, a *suite* de testes oficial para ECMAScript. Para este fim, introduzimos a ECMA-SL: uma linguagem intermédia dedicada à análise e especificação de ECMAScript. Apresentamos também ECMA-SL2English, uma ferramenta para geração da versão em inglês do documento HTML do *standard* ECMAScript a partir da ECMARef5. O documento HTML resultante é comparado com o documento oficial, sendo os resultados analisados detalhadamente. Em comparação com outros interpretadores de referência existentes, a nossa abordagem facilita a passagem de uma descrição textual do *standard* para uma executável.

**Keywords:** ECMAScript, Linguagem de especificação, Interpretadores de referência, Linguagens dinâmicas, Test262, OCaml

# Contents

# List of Tables

# List of Figures

# Chapter 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 StackOverflow[3]. ECMAScript is specified in the ECMAScript standard [1], 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. Figure 1.1 presents the evolution in number of pages of the official ECMAScript standard, by version. Note that the language standard suffered a large addition of pages between ES7 and ES8 and this occurred in a time interval of only one year. This shows how fast the language evolves.

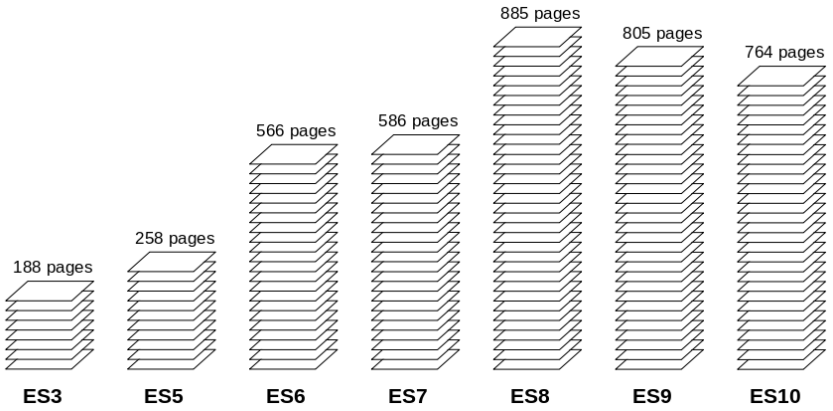



Figure 1.1: The evolution on the number of pages of the ECMAScript standard official document

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. Furthermore, 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,

[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

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 [2] 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 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.

**Contributions** Overall, the ECMA-SL project contributes to the research effort to streamline the management and maintenance of the ECMAScript standard by developing:

- ECMA-SL: a new intermediate language for the specification of the ECMAScript language. Our implementation of ECMA-SL includes an ECMA-SL parser, a compiler from ECMA-SL to Core ECMA-SL, and a Core ECMA-SL interpreter (implemented in the context of a parallel thesis [3]).
- ECMARef5: a reference interpreter of the ECMAScript standard (version 5) written in ECMA-SL. We have chosen to target the fifth version of the standard for two reasons. First, it the most stable version of the language, being entirely supported by all major browsers. Second, the size of the

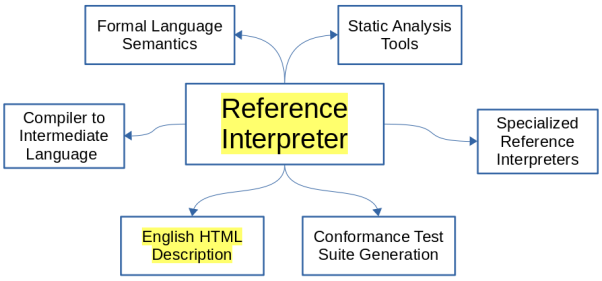



Figure 1.2: Tools that can be created from a ES5 reference interpreter

more recent versions of the standard would make it impossible to complete a reference interpreter within the time frame of this project. We must note, however, that there are various transpilers [4] from the more recent versions of the standard to `ECMAScript` 5.

- `ECMA-SL2English`: an HTML code generation tool that we use to automatically generate the English HTML version of the `ECMAScript` standard from the code of `ECMARef5`.

**Evaluation** We have thoroughly evaluated the two main outcomes of this thesis: the `ECMARef5` reference interpreter and the `ECMA-SL2English` HTML code generation tool. `ECMARef5` was tested against `Test262` [5], the official `ECMAScript` test suite. Out of a total of 12,068 applicable tests, we pass 99.6% of the tests, giving us a strong guarantee that `ECMARef5` implements the `ECMAScript` 5 standard correctly. We evaluated `ECMA-SL2English` by comparing the generated version of the standard against its official version. To this end, we made use of both classical text-based comparison metrics [6] and HTML-specific metrics based on the concept of tree similarity [7]. We have obtained consistently high scores for both classes of metrics, thereby validating our methodology for automatically generating the HTML version of the `ECMAScript` standard.

**Other applications** Figure 1.2 presents other tools that we believe can be generated from the code of `ECMARef5`; for example:

1. Compilers to intermediate languages [8; 9]: the idea is to first compile the input program to `Core ECMA-SL` by partially evaluating [10] `ECMARef5` on the input program and then compile the obtained `Core ECMA-SL` program to the targeted intermediate representation using a custom made compiler;

2. Automatic conformance test suite generators: the idea is to automatically synthesise a test suite for JavaScript engines by applying a dynamic symbolic execution tool, in the spirit of DART [11], to `ECMARef5`. Dynamic symbolic execution tools generate concrete inputs for the program to analyse by exploring multiple execution paths. In the case of `ECMARef5`, the generated concrete inputs would be the `ECMAScript` programs to be executed.

**Related work** There have been numerous research projects with the goal of developing a trustworthy reference implementation of the `ECMAScript` standard [12; 13; 14; 15]. Among these projects, the *JSCert* project is of special interest to us as it was the first project to emphasize the importance of linking the code of the reference interpreter to the text of the standard. The authors of JSCert claim that the code of the JSCert semantics is *eyeball-close* to the text of the standard, arguing that if one places side-by-side the English prose of the standard and the formal rules of JSCert and compares the two, *"one line of ES5 pseudo-code corresponds to one or two rules in JSCert"*. In this thesis, we improve on the

eyeball closeness methodology by taking the human out of the process. More concretely, using *ECMA-SL2English* we can precisely measure the closeness between our implementation of the standard and the official one through the use of out-of-the-box text-comparison metrics. In general, we believe that this project is a steppingstone into establishing more robust methodologies for implementing reference implementations of programming languages tightly connected to their respective standards.

**Thesis structure** This document is organized as follows: In Chapter 2, we give a detailed description of the `ECMAScript` 5 language focusing on its key aspects, such as built-in objects and internal functions. In Chapter 3, we give an overview of several other research efforts related to the work we performed in this thesis. In Chapter 4, we present both the `ECMA-SL` (4.1) and the `Core ECMA-SL` (4.2) intermediate languages, also describing the compilation process from `ECMA-SL` to `Core ECMA-SL`. In Chapter 5, we start by explaining the internal representations that we have used to model the different internal types of the `ECMAScript` standard (5.1), followed by the description of our implementation of `ECMAScript` built-in objects and initial heap (5.2). In this chapter, we also 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). In Chapter 6, we present `ECMA-SL2English`; we describe the HTML structure of the `ECMAScript` standard (6.1), present the main code generation algorithm (6.2) together with the annotations we provide to the code generator (6.3). In Chapter 7, we evaluate the main outcomes of this thesis: `ECMARef5` 7.1 and `ECMA-SL2English` 7.2. Finally, Chapter 8 draws some conclusions about our work and points out some future research directions.

# Chapter 2

# Background

## 2.1 ECMAScript Standard

The ECMAScript Standard edition 5.1 [1] is the official document that defines the ECMAScript scripting language (commonly known as JavaScript) 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.

The following subsections give a detailed description of the ES5 language based on the standard. We put the focus on the key aspects of the language, like built-in objects, semantics of some expressions and statements and internal functions.

### 2.1.1 Language Overview

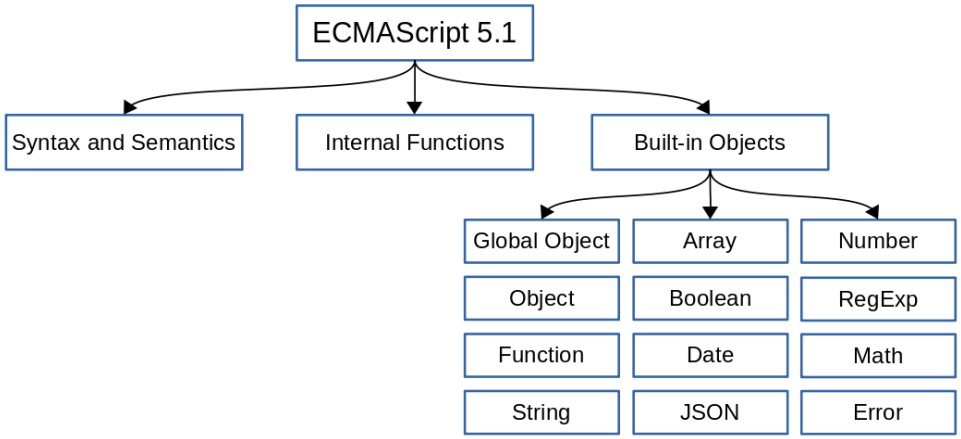



Figure 2.1: A graphical overview of the main components of the ECMAScript language version 5.1

We divide the ES5 language in three main components: syntax and semantics; internal functions; and built-in objects. This is illustrated in the Figure 2.1.

The *syntax and semantics* component groups all the syntactic grammars and semantics of expressions (assignment expressions, built-in operators, ...), statements (loop statements, conditional statements, ...), built-in types (*Undefined*, *Null*, *Boolean*, *Number*, *String*, and *Object*) 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 cannot call these functions directly. A detailed explanation of the internal functions is presented in 2.1.12.

The final component is the one that contains all the *built-in objects* available whenever an ES5 program executes. These built-in objects include 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*.

### 2.1.2 ES5 Objects and Properties

The ES5 language is object-based. This means that basic language and host facilities are provided by objects and an ES5 program is a cluster of communicating objects. An object is defined as a collection of properties. These properties are containers that hold other objects, primitive values, or functions. Each property contains attributes that determine how it is used. Each object property is either described as an internal property, a named data property, or a named accessor property. Objects are dynamic in that it is possible to add/remove properties to/from an object during execution.

#### Internal Properties vs Named Properties

*Internal properties* contain values that provide meta-information about the object they are associated with, for instance, the object type (e.g., `"Function"`, `"Array"`, `"String"`, etc...). They are meta-properties and cannot be directly accessed or modified by an ES5 program. They are used for the implementation of the internal algorithms and operations presented in the standard. Every ES5 object contains, at least, three internal properties:

1. `[[Class]]` representing the type of the object in the form of a string;
2. `[[Prototype]]` representing the internal prototype of the object (used to implement *prototype-based inheritance* discussed in 2.1.9). Its value is either a pointer to an object, also referred to as an object location, or `null`;
3. `[[Extensible]]` storing a boolean that determines whether or not it is possible to add new named properties to the object.

*Named properties* are the properties explicitly created by the program. They are split in two subtypes: *named data properties* and *named accessor properties*.

Figure 2.2 shows an ES5 program together with the object graph on which it operates. In line `1`, we declare the variable `car` as an object containing two named properties: `isMoving` and `color`. In 2-a), we show the object resulting from the evaluation of this assignment with its two named properties (rectangle in blue) and three default internal properties (red-dashed rectangle). The small blue rectangles represent *Property Descriptors* (discussed in 2.1.3) that contain the value of the respective property. In line `2`, we execute a delete expression that removes the property `color` from the object `car`. In 2-b), we show that the property `color` was indeed removed from object `car`. In line `3`, it is shown how the execution of the statement `Object.freeze(car);` affects the internal property `[[Extensible]]` of the object `car`, henceforth inhibiting any code from adding/removing named properties to/from the object `car`. In 2-c), we show the change in the internal property `[[Extensible]]`. Line `4` shows an example that will take no effect in the object `car`.

```
1  var car = { isMoving: true, color: "red" };
2  delete car.color;
3  Object.freeze(car);
4  car.color = "red";
```

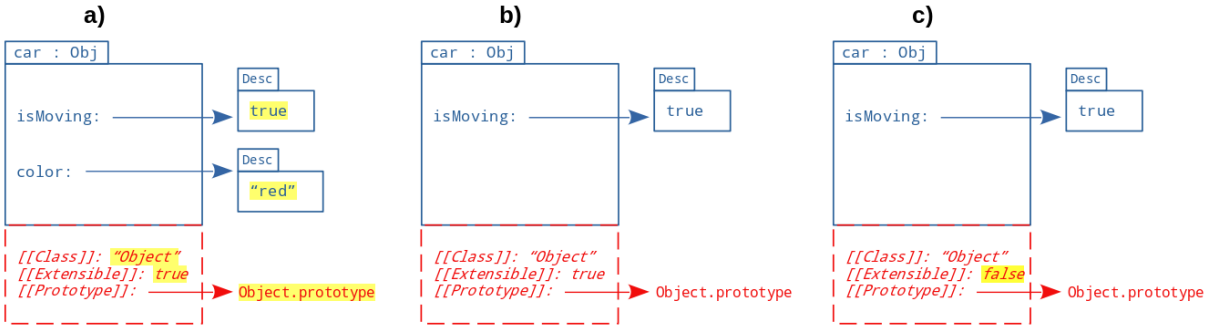



Figure 2.2: ES5 object and its properties

### 2.1.3 Property Descriptors

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:

1. Data Property Descriptors;
2. Accessor Property Descriptors; and
3. Generic Property Descriptors.

Below we describe each each type of property descriptor appealing to an example.

#### Data Property Descriptors

A data property descriptor holds a data value together with meta-information about the property. Each descriptor consists of a record with four attributes:

1. `[[Value]]` holds the actual property value that is of one of following language types: undefined, null, boolean, string, number, or object;
2. `[[Writable]]` determines whether the property value may or may not be modified;
3. `[[Enumerable]]` determines whether the property is to be visible by operations that iterate on properties of the object, like *for-in* enumerations;
4. `[[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.

Figure 2.3 shows the internal representation of ES5 data property descriptors. Note that the value of the property `color` is stored inside the `[[Value]]` attribute of the corresponding descriptor.

In ES5, a program can update existing properties or create new properties, either using the usual property assignment expression or the less usual built-in function `Object.defineProperty`. This built-in function lets the program specify the attributes associated with the given property descriptor. Depending on the way a property is defined, the attributes `[[Writable]]`, `[[Enumerable]]`, and `[[Configurable]]` of the corresponding descriptor have different values. There are two possible scenarios:

```
1  var car = { isMoving: true, color: "red" };
```

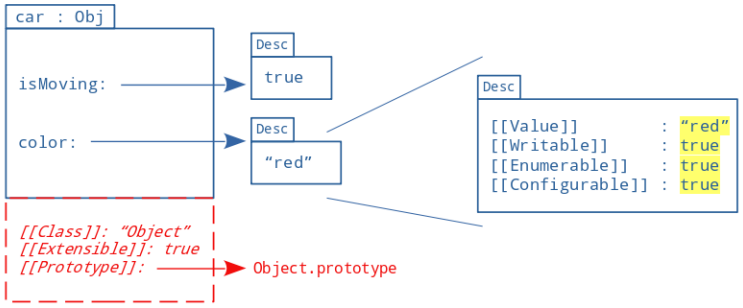



Figure 2.3: ES5 object and one of its data property descriptors

1. When using object initialiser expression or property assignment expression, all these attributes will have the value `true`;
2. When using the built-in function `Object.defineProperty`, if any of these is not specified it gets the default value of `false`.

### Accessor Property Descriptors

An accessor property descriptor associates a given property with a function *get* that computes its value and a function *set* that updates or sets its value. Each descriptor consists of a record with four attributes:

1. `[[Get]]` if defined, returns the property value for every get access that is performed on the property;
2. `[[Set]]` if defined, updates or sets a new property value for every set access that is performed on the property;
3. `[[Enumerable]]` and `[[Configurable]]` have the same meaning as the ones described above in *Data Property Descriptors*.

```
1  var car = { gas: 20,
2    get tank () { return this.gas > 20 ? "half-full" : "half-empty"; }, // getter
3    set tank (quant) { if (quant >= 0) { this.gas = quant; } } // setter
4  };
```

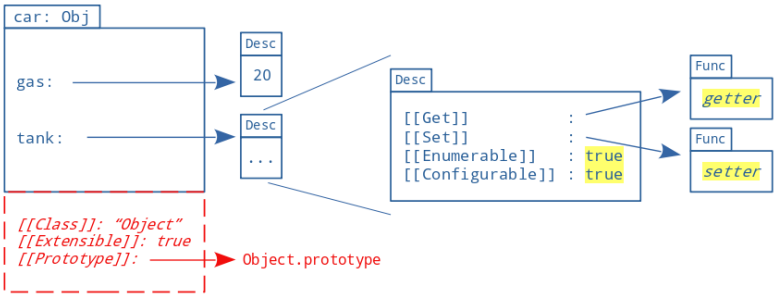



Figure 2.4: ES5 object and one of its accessor property descriptors

Figure 2.4 shows the internal representation of ES5 accessor property descriptors. Two functions with the same name, i.e. `tank`, are defined at the time of the creation of the object `car`, but with slightly different signatures: one is preceded with the keyword `get` and has no arguments; another is preceded

with the keyword `set` and has one argument, i.e., `quant`. Internally these functions are associated with two different attributes of the corresponding accessor property descriptor: `[[Get]]` and `[[Set]]`.

Analogously to data property descriptors, the value assigned to the attributes `[[Enumerable]]` and `[[Configurable]]` of an accessor property descriptor varies depending on how the property is updated or set. The attributes `[[Get]]` and `[[Set]]` are set to `undefined` when not specified.

```
1 car.tank; // "half-empty"
2 car.tank = 30;
3 car.tank; // "half-full"
4 car.tank = -10;
5 car.tank; // "half-full"
```

Figure 2.5: Program using the accessor property descriptor created in figure 2.4

Below we give a detailed description of the program given in Figure 2.5:

- in line `1`, we access the value of the property `tank`. The function *get* is executed returning the string `"half-empty"`, since the value of the property `gas` is not greater than 20;
- in line `2`, we set the value of the property `tank` to `30`. This triggers the execution of the *set* function that sets the value of the property `gas` to 30;
- in line `3`, we access again the value of `tank` which this time returns the string `"half-full"`;
- in line `4`, we try to set the value of the property `tank` to `-10`, that is an invalid value considering the code of the *set* function.
- in line `5`, we confirm that the execution of the *set* function did not change the state of the object `car`. We access the value of the property `tank` which results in the string `"half-full"`.

#### Generic Property Descriptors

A generic property descriptor is one that is neither a data nor an accessor property descriptor, but it can have two of the attributes identified in the other two descriptors: `[[Enumerable]]` and `[[Configurable]]`. This kind of property descriptor is only useful in the context of the internal algorithms and operations defined in the standard and they cannot be created by an ES5 program.

### 2.1.4 Function Objects

In ES5, *functions* are internally represented as *Function Objects* that besides the three internal properties contained by all ES5 objects, contain three extra internal properties:

1. `[[Scope]]` stores the scope chain in which the function was created (discussed later in 2.1.10);
2. `[[FormalParameters]]` contains the list of the function's formal parameters;
3. `[[Code]]` contains the code of the body of the function.

The remainder internal properties have the following values:

1. `[[Class]]` - all function objects have class `"Function"`;
2. `[[Prototype]]` - all function objects have prototype `Function.prototype`;
3. `[[Extensible]]` - all function objects are extensible (value `true`), meaning that they can be extended with new data properties.

Function objects also contain a single data property named *length* that indicates the number of the expected arguments of the corresponding function. This property is non-enumerable and immutable: it is read-only and non-configurable.

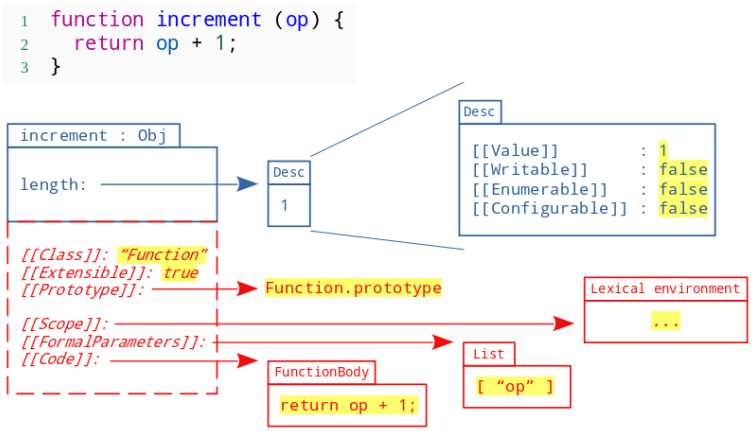



Figure 2.6: Function object and its properties

Figure 2.6 shows the code and the graphical representation of a Function object called `increment`. The internal properties with their corresponding values are presented in the red-dashed rectangle. Apart from these internal properties, it is possible to verify how the named data property `length` is represented including its corresponding attributes and values.

### 2.1.5 String Objects

In ES5, Strings may be represented either as String literals or String objects. In the latter case, String objects are wrappers around the primitive type string. The corresponding string value is maintained in an internal property called `[[PrimitiveValue]]`.

The three default object properties have the following values:

1. `[[Class]]` - all string objects have type `"String"`;
2. `[[Prototype]]` - all string objects have prototype `String.prototype`;
3. `[[Extensible]]` - all string objects are extensible (value `true`), meaning that they can be extended with new data properties.

String objects contain a data property named `length` that indicates the total number of characters of the corresponding primitive value. This property is also non-enumerable and immutable.

Figure 2.7 shows a program that creates a String object storing the primitive string `"Figures.odp"`, as well as its internal representation. The internal properties with their corresponding values are presented in the red-dashed rectangle. As referred above, the string value is presented in the internal property `[[PrimitiveValue]]`. Note that we also give the internal representation of the data property descriptor associated with the property `length`.

String objects differ from other ES5 objects in that they contain *indexing properties*. These properties are also represented by data property descriptors and are read-only, enumerable and non-configurable. The number of indexing properties in a String object are limited to the number of characters that form the containing string value (corresponding to the value of the *length* named data property), i.e., the example above shows that the string value contains 11 characters, so the String object has 11 indexing

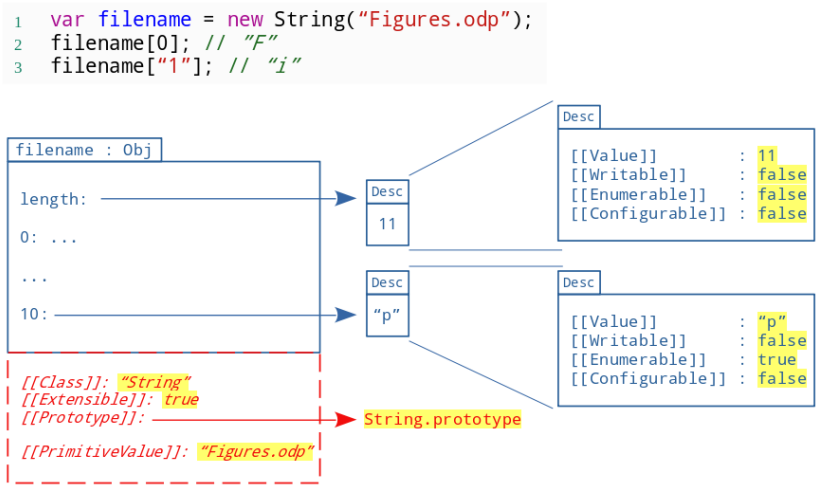



Figure 2.7: String object with its properties

properties, the first one, with index `0`, having the value `"F"` and the last one, with index `10`, having the value `"p"`. The access of an invalid indexing property, e.g., `11`, evaluates to the value `undefined`.

To access the value of these indexing properties, an ES5 program should contain expressions that enclose a valid integer value (or the equivalent string, e.g., `"1"`) in square brackets, as shown in the lines `2` and `3` in Figure 2.7.

### 2.1.6 Array Objects

In ES5, Arrays are internally represented as objects. Each array object only contains the three internal properties that any other ES5 object contains, which have the following values:

1. `[[Class]]` - all array objects have type `"Array"`;
2. `[[Prototype]]` - all array objects have prototype `Array.prototype`;
3. `[[Extensible]]` - all array objects are extensible (value `true`), meaning that they can be extended with new data properties.

These objects are created containing a data property named `length` that indicates the number of existing items in the array. This property is writable, non-enumerable and non-configurable.

Figure 2.8 shows the code and the graphical representation of an Array object called `cars`. The internal properties with their corresponding values are presented in the red-dashed rectangle. Apart from these internal properties, it is possible to see how the named data property `length` and the indexing properties are represented as well as their corresponding attributes and values.

Like String objects, Array objects also contain indexing properties, which are represented by data property descriptors. Unlike the indexing properties of String objects, the indexing properties of Array objects are writable, enumerable, and configurable. The number of indexing properties and the value of the property `length` are equal and always kept in sync, i.e., whenever a new indexing property is added to the array object, the `length` property is updated to match the new number of indexing properties; also, whenever the `length` property is changed, the indexing properties are also updated, either by removing some of these properties to match a lower `length` value or by adding new properties (with default value of `undefined`) to match a greater `length` value.

Figure 2.8 also represents the indexing properties of the Array `cars`, showing the data descriptor associated with index `2`. To access the value of these indexing properties, an ES5 program should

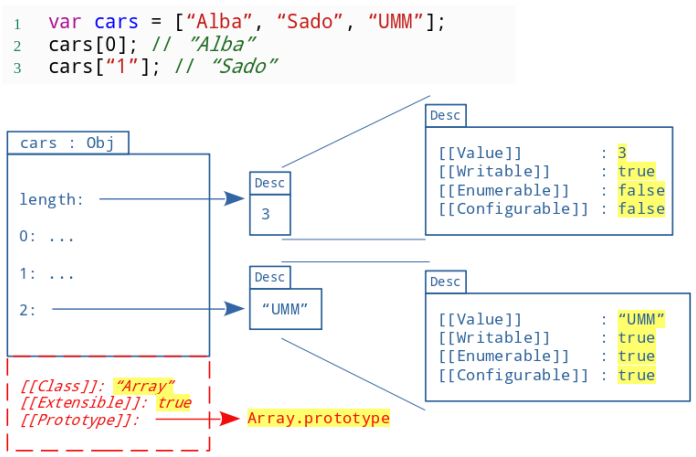



Figure 2.8: Array object with its properties

contain expressions that enclose a valid integer value (or the equivalent string, e.g., `"1"`) in square brackets, as shown in the lines 2 and 3 in the figure.

```
1  var a = [1, 2, 3];
2  Object.defineProperty(a, 1, { value: 4, configurable: false });
3  a.length = 0;
4  a; // [1, 4]
```

Figure 2.9: Example defining an array element to be non-configurable and trying to delete the contents of the array

In contrast to what happens with the named data property `length` of a String object, the one of an Array object is not immutable meaning that it is possible to add/remove items to/from the array by updating the value of `length`. For instance, if we set the length of `cars` to `0`, all the elements of the array are implicitly removed. Consider the example in Figure 2.9. In line `1`, we declare an array containing three elements: the numbers `1`, `2`, and `3`. In line `2`, we modify the element in the index `1` to the number `4`, making this index property non-configurable. In line `3`, we set the value of the `length` property to `0`, trying to implicitly remove the contents of the array. In line `4`, we check the contents of the array and verify that it still contains elements, in this case, numbers `1` and `4`. This happened due to the modification in line `2`: a non-configurable property cannot be deleted. This causes the algorithm to stop removing the elements at this index. The result is that the code in line `3` removed only one element.

### 2.1.7 Global Object

In ES5, the *Global Object* is the object that holds global variables, global functions, and all the built-in objects. For instance, when a global variable is created as part of an executing program, it is stored as a property of the Global Object. Considering any of the worldwide most used desktop web browsers[1], one is able to access the Global Object using the global variable `window` or, at the global scope level, the keyword `this`.

In Figure 2.10, we show a program that declares two global variables `x` and `y`, in lines `1` and `2`, setting them respectively to the values `3` and `4`. In line `3`, we update the value of the property `x` of the Global Object to `10` but through the use of the global variable `window`. In line `4`, we update the value of `y` to `10` using another syntax: with the keyword `this`, that in this global scope context is resolved to the Global

[1] Desktop Browser Market Share Worldwide (July 2021) - https://gs.statcounter.com/browser-market-share/desktop/worldwide

```
1  var x = 3;
2  var y = 4;
3  window.x = 10;
4  this["y"] = 10;
5  x + y; // 20
```

Figure 2.10: Accessing properties of the *Global Object*

object, and accessing the property using array-like notation. Finally, in line 5 we evaluate an addition of these two global variables, resulting the value `20`.

### 2.1.8 Other Objects

As stated above in 2.1.1, ES5 comes with a large number of built-in objects, including the String object or the Array object. In the above subsections, we have described the Global object, the String object, the Array object, and the Function object. Besides these objects, there are the Object object, the Number object, the Boolean object, the Math object, the Date object, the RegExp object, the Error object, and the JSON object. Here, we describe the Number object and the Boolean object for illustrative purposes. The other objects do not pose additional challenges.

Number objects and Boolean objects are wrappers around primitive numbers and booleans, respectively. Both these objects store their corresponding primitive values in the `[[PrimitiveValue]]` internal property. The three default internal properties of Number objects and Boolean objects are given in the following table:

| Internal Property | Number Objects | Boolean Objects |
|---|---|---|
| `[[Class]]` | `"Number"` | `"Boolean"` |
| `[[Prototype]]` | `Number.prototype` | `Boolean.prototype` |
| `[[Extensible]]` | `true` | `true` |

Table 2.1: Default values for the internal properties of any Number object or Boolean object

All Number objects have the same prototype, `Number.prototype`, which stores the methods shared by all the Number objects. For instance, the method `toFixed` formats a number using fixed-point notation and is accessible to all objects of type Number. This method receives as argument the number of digits after the decimal point. Consider the code in Figure 2.11. In lines `1` and `2`, we declare two variables

```
1  var numLit = 12;
2  var numObj = new Number(5.678);
3  numLit.toFixed(2); // "12.00"
4  numObj.toFixed(1); // "5.7"
```

Figure 2.11: Using the method `toFixed` accessible to all Number objects

`numLit` and `numObj`, setting them respectively to `12` and `5.678`. Note that we are using two different syntactic constructs to create two numbers. In the case of line `1` we are assigning the variable `numLit` the primitive number `12`. In line `2` we are assigning the variable `numObj` a Number object wrapping the primitive number `5.678`. Also note that in ES5 both integers and real numbers are represented with the same type: Number.

In lines `3` and `4`, we use the method `toFixed` to get the string representations of both number variables. In line `3`, we pass the value `2` as argument to this method and get a string containing the number

`12` with the fractional part padded with two zeros: the string `"12.00"`. In line `4`, we pass the value `1` as argument getting the string `"5.7"`. Note that the only digit after the decimal point is not `6`, as one may expect, but `7`. This happens because this method `toFixed` rounds the number so that it has the specified length.

### 2.1.9 ES5 Prototype-based Inheritance

ES5 is a class-less language. The distinction between classes and instances present in other object-oriented languages (like Java) does not exist in ES5. An object is always an instance and it inherits properties from other objects through the use of the internal property `[[Prototype]]`. This is called *Prototype-based Inheritance* [8]. When an ES5 object is created, it takes an existing object as its prototype. That object also has its own prototype defined. These together form a *prototype chain*.

Every time a program wants to retrieve a value of an object's property, that property is searched in the object itself. When not present, the prototype chain is traversed until the property is found or until the search reaches the top-level object which has no prototype set.

```
1  var tesla = { brand: "Tesla" };
2  var modelS = Object.create(tesla, { model: { value: "Model S" } });
3  var modelX = Object.create(tesla, { model: { value: "Model X" } });
4  modelS.brand;
5  modelX.color;
```

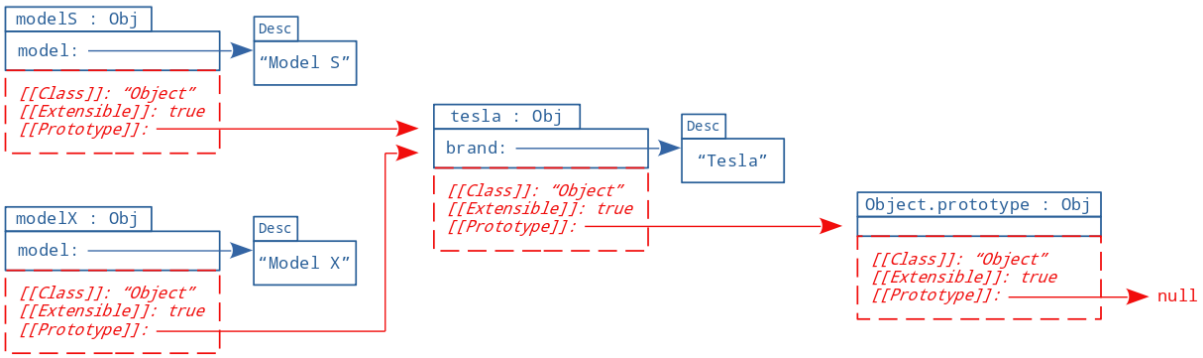



Figure 2.12: Program demonstrating ES5 prototype-based inheritance (top); graphical illustration of the prototype chain obtained after executing the program (bottom)

Figure 2.12 shows a program that illustrates the inner-workings of ES5 prototype-based inheritance. In line `1`, the program sets the variable `tesla` to an object that contains a single property `brand` with value `"Tesla"`. By default, the prototype of this object is set to `Object.prototype`. In lines `2` and `3`, the program makes use of the method `Object.create` to create two new objects, respectively assigned to variables `modelS` and `modelX`. This method has two parameters: the first is the object to be set as the prototype of the new object; and the second is an object containing the named properties that are going to be added to the new object. So, both `modelS` and `modelX` have prototype `tesla` and have a single property each, `model`, with the values `"Model S"` and `"Model X"`, respectively. In line `4`, the program tries to get the value of the property `brand` that does not exist in the object `modelS`. However, the property `brand` exists in the prototype chain of `modelS`, so the property lookup evaluates to the string `"Tesla"`. Finally, in line `5`, the program tries to get the value of the property `color` which does neither exist on the object `modelX` nor in any of the objects of its prototype chain. In this case, the value returned is `undefined`.

### 2.1.10 ES5 Functions and Scoping

In ES5, code is run in three different types of *execution context*: globally, inside functions, and during the execution of the `eval` built-in function. Global code is the code that is at the top-level of an ES5 program and not inside functions.

Each execution context contains a *Lexical Environment*. A Lexical Environment is a type defined in the ES5 standard for storing the bindings of variables and function identifiers. It can be seen as a pair that contains an *Environment Record* and a (possibly null) reference to an outer Lexical Environment. An Environment Record is an internal object, created upon the invocation of a function, that maps the variables declared in the body of that function and its parameters to their respective values. It can be seen as a lookup table of key-value pairs that maps identifiers to their corresponding values.

```
1  var makeIdGen = function (prefix) {
2    var count = 0;
3    var getId = function () {
4      return prefix + "_id_" + (count++)
5    };
6    var reset = function () { count = 0 };
7
8    return { getId: getId, reset: reset }
9  }
10 var ig1 = makeIdGen("foo");
11 var id1 = ig1.getId();
12 ig1.reset();
```

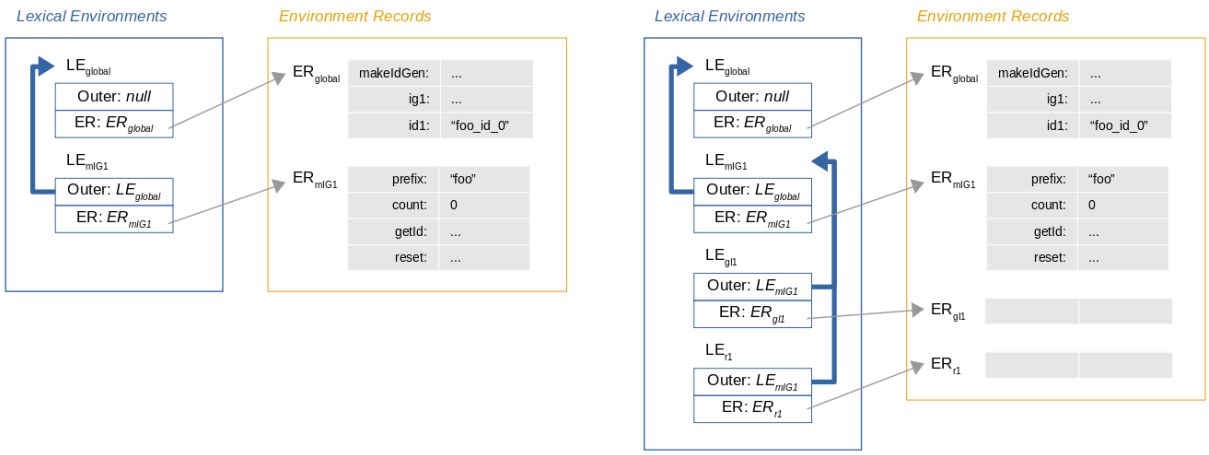



Figure 2.13: Identifier generator program (top); Lexical Environments and Environment Records that result from the execution of the program until line 10 (bottom left); Lexical Environments and Environment Records that result from the complete execution of the program (bottom right)

Scope resolution is performed by inspecting the identifier bindings present in lexical environments. Consider the identifier (ID) generator example presented in Figure 2.13. The function `makeIdGen` takes a string `prefix`, and returns a new object with two properties: `getId`, storing a function for creating fresh IDs; and `reset`, storing a function that resets the ID generator. `getId` guarantees that the returned ID is fresh by using a counter, stored in variable `count`, which is appended to the generated ID string (`prefix + "_id_"`) and is incremented afterwards. Note that the code inside the functions `getId` and `reset` accesses variables that are not defined inside these functions, though the program executes normally.

During the execution of the code in the figure, various lexical environments and associated environment records are created. On the left-side of the figure, we have a representation of the lexical environments (in the blue rectangle) and environment records (in the yellow rectangle) created by the execution of the program until line `10`. On the right-side, we represent the lexical environments and

environments records created as a result of the complete execution of the program.

The $\texttt{LE}_{\texttt{global}}$ is the lexical environment created for the global scope. It contains a pointer to the environment record $\texttt{ER}_{\texttt{global}}$ with the mappings for the global variables `ig1` and `id1` and the global function `makeIdGen`. Note that $\texttt{ER}_{\texttt{global}}$ is the global object discussed in Section 2.1.7, which is accessible via the `window` variable and the `this` keyword at the global scope level. Hence, it is possible to change the bindings of global variables by interacting directly with the global object.

In line `10`, the program creates a new ID generator and assigns it to the global variable `ig1`. When calling the function `makeIdGen` with the argument `"foo"`, the lexical environment $\texttt{LE}_{\texttt{mIG1}}$ is created with mappings for the variables `prefix`, `count`, `getId`, and `reset`, respectively set to `"foo"`, `0` and two different anonymous functions with no arguments (lines `3` till `6`). The lexical environment $\texttt{LE}_{\texttt{mIG1}}$ points to lexical environment $\texttt{LE}_{\texttt{global}}$ because the function `makeIdGen` is defined in the global context.

In line `11`, the program uses the generator `ig1` to obtain a fresh ID and assigns it to the global variable `id1`. When calling the function `getId`, the lexical environment $\texttt{LE}_{\texttt{gI1}}$ is created with an empty mapping. This lexical environment points to lexical environment $\texttt{LE}_{\texttt{mIG1}}$ because the function `getId` is defined inside `makeIdGen`.

During the execution of `getId`, variables are resolved with respect to the current list of lexical environments ([ $\texttt{LE}_{\texttt{gI1}}$, $\texttt{LE}_{\texttt{mIG1}}$, $\texttt{LE}_{\texttt{global}}$ ]), traversing this list from left to right. Such lists are called *scope chains*. When the `prefix` variable is accessed, internally ES5 tries to get its value by searching the environment record of the current lexical environment, $\texttt{ER}_{\texttt{gI1}}$. As no match is found, the search proceeds to the next lexical environment in the scope chain, in this case, the lexical environment $\texttt{LE}_{\texttt{mIG1}}$. When searching for `prefix` in the environment record $\texttt{ER}_{\texttt{mIG1}}$, a mapping is found and its associated value is returned. The same applies for the variable `count`.

Finally, the programs calls the `reset` function to reset the ID generator `ig1`. During the execution of `reset`, the programs assigns `count` to `0`. To this end, the ES5 semantics will first have to find the environment record that defines `count` in the current scope chain. The traversal of the scope chain works as explained for `getID`, meaning that the semantics will find `count` in the environment record $\texttt{ER}_{\texttt{mIG1}}$. Once the corresponding environment record is found, the semantics will simply update the value of `count` appropriately.

### 2.1.11 ES5 Syntax and Control Flow

The purpose of this section is to illustrate how the standard specifies the semantics of the various constructs of the language. To this end, we give a detailed description of the specification of the *If-Then-Else* statement and the variable assignment expression.

#### References

Before proceeding to the description of the semantics of the If-Then-Else and variable assignment, we must introduce ES5 *references*. A reference represents an unresolved name binding. It consists of a *base value* and a *referenced name*. The base value is either `undefined`, an `object`, a `boolean`, a `string`, a `number`, or an environment record. A base value of `undefined` indicates that the reference could not be resolved to a binding. The referenced name is a string, typically the name of the binding we want to resolve, e.g., the name of a variable or the name of a property of an object. For instance, $(\texttt{ER}_{\texttt{f}}, \texttt{x})$ denotes a variable reference, where `x` is the variable name and $\texttt{ER}_{\texttt{f}}$ the environment record where `x` is to be found, and $(\texttt{Obj}_{\texttt{a}}, \texttt{p})$ denotes an object reference where `p` is the name of the property name and $\texttt{Obj}_{\texttt{a}}$ the object in whose prototype chain `p` is to be found. Note that the base value of an object reference is not necessarily the object where the referenced property is defined, but rather an object in

**Semantics**

The production *IfStatement* : **if** ( *Expression* ) *Statement* **else** *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. If ToBoolean(GetValue(*exprRef*)) is **true**, then
    a. Return the result of evaluating the first *Statement*.
3. Else,
    a. Return the result of evaluating the second *Statement*.

Figure 2.14: Semantics of the If-Then-Else statement defined in the standard

whose prototype chain that property can be found.

To obtain the associated value, the reference needs to be dereferenced. This is the job of the internal function `GetValue`.

### If-Then-Else

In Figure 2.14, we show the fragment of the ES5 standard that specifies the semantics of the If-Then-Else statement. This statement is described in three steps: the evaluation of the guard expression (line `1`), returning the reference `exprRef`; the conversion of this reference to a boolean value using the internal functions `ToBoolean` and `GetValue` (line `2`); and the evaluation of one of the two statements (then or else) depending on whether the value returned is `true` or `false`.

The semantics of the If-Then-Else statement is exactly what one would expect, but for the detail of the call to the internal `ToBoolean` function. This function is called because the resolution of `exprRef` to a value (using the internal function `GetValue`) may yield a non-boolean value. The `ToBoolean` function will then perform the necessary coercion by considering both the type of its argument and its value. For instance, consider the following program:

```
var car = "";
if(car) { 1 } else { 0 }
```

The evaluation of the If-Then-Else statement will generate the value `0` because the internal function `ToBoolean` returns `false` when supplied the empty string. For any other string value, it returns `true`. Implicit coercions (such as those performed by `ToBoolean`) are a common source of mistakes in ES5 programs.

### Variable Assignment

In ES5, the basic assignment operator is the equal operator, `=`, which assigns the value of its right operand to its left operand. That is, `x = y` assigns the value of `y` to `x`. The standard also defines other assignment operators, that are shorthand for other standard operations. For instance, the multiplication assignment operator, `*=`, where `x *= y` is syntactic sugar for `x = x * y`. Figure 2.15 shows the semantics of the simple assignment defined in the ES5 standard. Consider the following expression: `x = y`. We can describe the semantics of this expression in four different steps: evaluate the variable `x` (line `1`), assigning the returned reference to `lref`; evaluate the variable `y` (line `2`), assigning the returned reference to `rref`; get the value associated with `rref`, using the internal function `GetValue`, and assigning it to `rval`; assign this value to the reference `lref` using the internal function `PutValue`. This will replace any previous value assigned to `x` with the value of `rval`. Note that some assertions are checked on the `lref` before calling `PutValue` (line 4) to make sure that some reserved keywords (`eval` and `arguments`) are not used on the left-side of the assignment expression.

**11.13.1 Simple Assignment ( = )**

The production *AssignmentExpression* : *LeftHandSideExpression* = *AssignmentExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *rref* be the result of evaluating *AssignmentExpression*.
3. Let *rval* be GetValue(*rref*).
4. Throw a **SyntaxError** exception if the following conditions are all true:
   - Type(*lref*) is Reference is **true**
   - IsStrictReference(*lref*) is **true**
   - Type(GetBase(*lref*)) is Environment Record
   - GetReferencedName(*lref*) is either `"eval"` or `"arguments"`
5. Call PutValue(*lref*, *rval*).
6. Return *rval*.

Figure 2.15: Semantics of a simple assignment expression defined in the standard

### 2.1.12 ES5 Internal Functions

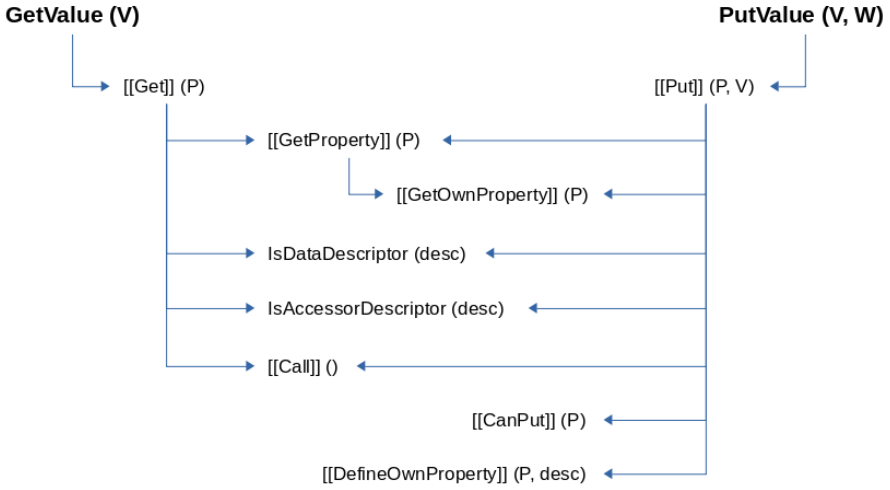
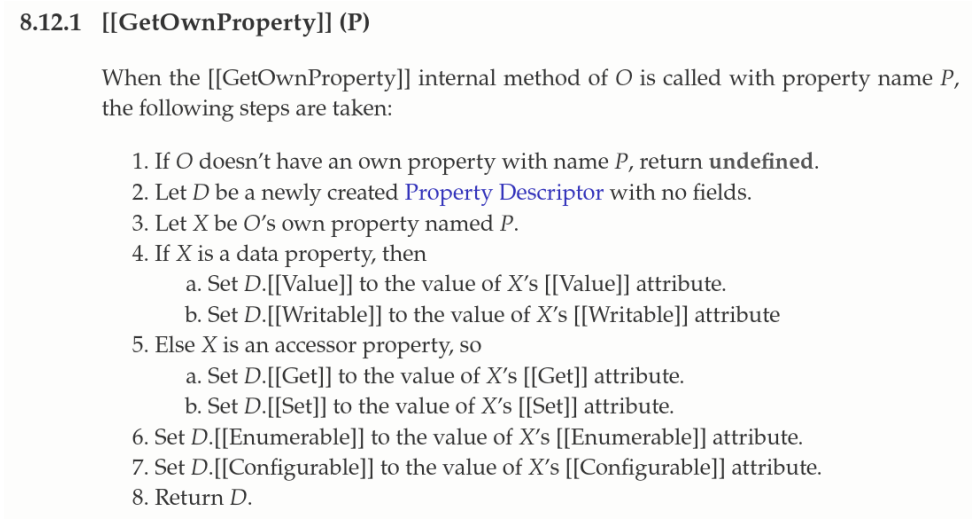



Figure 2.16: A portion of the Call graphs of the internal functions `GetValue` and `PutValue`

For the purpose of describing the semantics of the ES5 language, the standard defines a number of internal functions with corresponding algorithms. The range of internal functions goes from type conversion and testing, to objects' internal methods, functions to operate on property descriptors, or to operate or access the components of references.

In the previous subsection, we mentioned two internal functions (`GetValue` and `PutValue`) used to deal with references. `GetValue` resolves a reference to a value, that is, the value of a property of an object or the value bound to an identifier present in an environment record. `PutValue` assigns a certain value to a variable or an object property pointed to by a reference. Both these functions have complex behaviours. Their definitions are intertwined with other internal functions, making it difficult to understand all their possible behaviours. Figure 2.16 shows a portion of the call graphs of the `GetValue` and the `PutValue` internal functions. We only present the internal functions that deal with objects and property descriptors.

Next, we will give a more detailed explanation of two of these internal functions called as part of the execution of `GetValue` and `PutValue`, respectively `[[GetOwnProperty]]` and `[[GetProperty]]`. In the following we consider the application of these functions to an object `O` and property `P`.



**8.12.1 [[GetOwnProperty]] (P)**

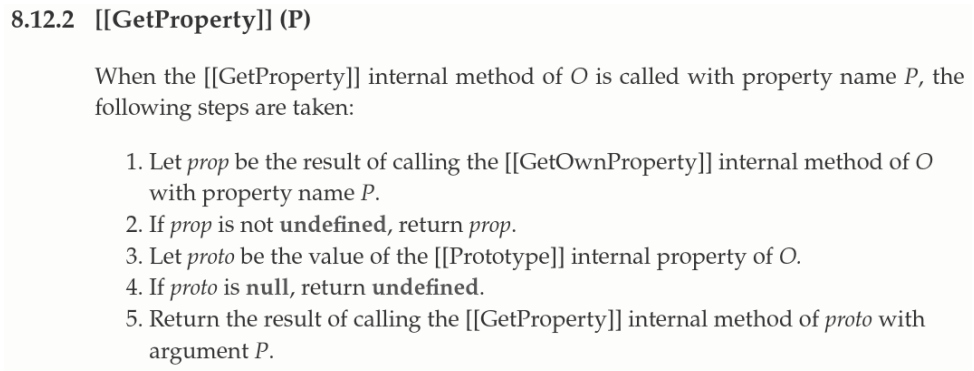
When the [[GetOwnProperty]] internal method of *O* is called with property name *P*, the following steps are taken:

1. If *O* doesn't have an own property with name *P*, return **undefined**.
2. Let *D* be a newly created Property Descriptor with no fields.
3. Let *X* be *O*'s own property named *P*.
4. If *X* is a data property, then
   a. Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.
   b. Set *D*.[[Writable]] to the value of *X*'s [[Writable]] attribute
5. Else *X* is an accessor property, so
   a. Set *D*.[[Get]] to the value of *X*'s [[Get]] attribute.
   b. Set *D*.[[Set]] to the value of *X*'s [[Set]] attribute.
6. Set *D*.[[Enumerable]] to the value of *X*'s [[Enumerable]] attribute.
7. Set *D*.[[Configurable]] to the value of *X*'s [[Configurable]] attribute.
8. Return *D*.

Figure 2.17: The specification of the Object internal function `[[GetOwnProperty]]`

### GetOwnProperty

The internal function `[[GetOwnProperty]]` retrieves a property descriptor representing the value of the property `P` in the object `O` or `undefined` if `P` does not exist in `O` (line `1`). The section of the standard that specifies this function is given in Figure 2.17. Note that instead of returning the property descriptor obtained in line `3`, this internal function returns a copy of that property descriptor (lines `4` till `8`). In this way, it is guaranteed that further changes to the returning property descriptor do not affect the one stored in the object `O`.



**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*.

Figure 2.18: The specification of the Object internal function `[[GetProperty]]`

### GetProperty

The internal function `[[GetProperty]]` recursively traverses the prototype chain of the object `O` to retrieve the descriptor associated with the property `P`. The semantics of this function is presented in Figure 2.18. Initially, this function calls `[[GetOwnProperty]]` to retrieve this descriptor from the object `O` (line `1`). If not found, it gets the prototype of the object (line `3`) and recursively calls itself on this prototype object (line `5`). This is performed until the descriptor is obtained or until the end of the prototype chain is reached, in which case it returns `undefined` (line `4`).

# Chapter 3

# Related Work

The research literature covers a wide range of program analysis and instrumentation techniques for ECMAScript, such as: type systems [16; 17], abstract interpreters [18], points-to analyses [19], program logics [8; 20], operational semantics [13; 12; 14], intermediate representations/compilers [8], among others. Here we focus on operational semantics/reference interpreters for ECMAScript and intermediate languages and compilers for ECMAScript analyses.

**Operational semantics and reference interpreters for ECMAScript** 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 in chronological order.

Maffeis et al. [13] were the first to design an operational semantics for ECMAScript following the standard faithfully. They designed a small-step operational semantics for the third version of the standard and used this semantics to reason about the security properties of web applications and mashups [21; 22]. However, they did not mechanise their semantics, leaving it as a long text document comprising a large number of semantic rules written in their own custom-made syntax.

Guha et al. [9] defined $\lambda_{JS}$, a core lambda calculus that captures the most fundamental features of ECMAScript 3: extensible objects, prototype-based inheritance, and dynamic function calls. $\lambda_{JS}$ comes with a *Racket* [23] interpreter and a de-sugaring translation that converts ES3 programs into $\lambda_{JS}$ expressions. This project also includes a type system for checking a simple confinement property of $\lambda_{JS}$ programs. Importantly, $\lambda_{JS}$ does not support dynamic code evaluation via `eval` and several of the language built-in libraries. Later, Politz et al. [24] extended $\lambda_{JS}$ from ES3 to ES5 including, for the first time, a formal semantics of ECMAScript accessor property descriptors (a.k.a. getters and setters) and a thorough treatment of the `eval` statement.

Bodin et al. [12] developed *JSCert*, the first mechanised specification of the ECMAScript semantics. The authors formalised a pretty-big-step semantics [25] of ECMAScript 5 in the *Coq* [26] 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* [5]. Later, Gardner et al. [27] extended the JSRef reference interpreter with support for ES5 Arrays. To this end, the authors linked JSRef to the *Google's V8* [28] 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 methodology of eye-ball closeness. We improve

on this methodology 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.

Park et al. presented *KJS* [14], the most complete formal semantics of ES5 to this day. KJS was developed using the *K framework* [29], a well established term-rewriting system supporting various types of symbolic analyses. KJS was thoroughly tested against Test262 passing a total of 2,782 core language tests. The K framework comes with a built-in symbolic analysis mechanism based on reachability logic [30]. This means that if one formalises the semantics of a given programming language in the K framework, one obtains a symbolic analysis for that language for free. Taking advantage of this mechanism, the authors of KJS later leveraged their ECMAScript semantics implemented in K to symbolically reason about simple ECMAScript programs [31].

Charguéraud et al. presented *JSExplain* [15], a reference interpreter for the ECMAScript 5 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.

Besides complete implementations of the ECMAScript standard, the research community has also worked on stand-alone reference implementations of some of the built-in libraries that comprise the ECMAScript standard. For instance, Sampaio et al. [32] developed a trusted infrastructure that includes a reference implementation of JavaScript Promises following the standard line-by-line. The authors also developed JaVerT.Click, a symbolic execution tool that, for the first time, supported reasoning about JavaScript programs that use promises.

**Intermediate representations and compilers** The complexity of the ECMAScript language, renders the direct development of precise program analysis and verification tools for ECMAScript an extremely challenging task. A well established approach to deal with this complexity, is to move the analysis to simpler intermediate languages via compilation. Instead of analysing an ECMAScript program directly, one first compiles it to a simple intermediate language and applies the analysis at the intermediate language level. Hence, the research community has proposed a wide variety of intermediate representations for ECMAScript analysis. These intermediate representations can be divided into two main classes: (1) those that are aimed at syntax-directed analysis, which are generally variations of either imperative-style While languages or lambda calculi, such as $\lambda_{JS}$ [9], *S5* [24], *notJS* [33] , and *JISET* [34]; and (2) those that are aimed at control-flow-graph-based analysis, which are generally variations of simple goto languages, such as the IR of *TAJS* [16; 35] , *WALA* [36] , *JSIR* [37] , and *JSIL* [8]. In both cases, typical intermediate languages for ECMAScript analysis have native support for extensible objects and their associated operations. Importantly, the first class of intermediate representations is more suited for high-level analyses, such as type-checking and type inference, while the second class is more suited for low-level analyses, such as symbolic execution and bounded model-checking .

In the following, we describe the two intermediate representations that we consider to be the closest to ECMA-SL: JSIL, the intermediate language of *JaVerT* [8; 38; 39], and JISET, the intermediate representation that is at the core of a novel ECMAScript implementation developed at *KAIST* [34].

Fragoso Santos et al. proposed JSIL [8], a simple goto language with support for extensible objects,

dynamic function calls, and dynamic code evaluation. JSIL was developed in the context of the *JaVerT* project. JaVerT is a separation-logic-based verification tool for reasoning about functional correctness properties of ES5 (strict) programs. JaVerT works by first compiling the given `ECMAScript` program to JSIL using a compiler called JS-2-JSIL. Verification takes place at the JSIL level using a dedicated verification engine for JSIL called *JSILVerify*. Importantly, JaVerT does not apply any simplifications to the `ECMAScript` semantics, analysing all the corner-cases of the language.

Very recently, Jihyeok Park et al. presented *JISET* [34], a JavaScript IR-based semantics extraction toolchain with which the authors were able to semi-automatically obtain a partial implementation of `ECMAScript` 10. More concretely, the authors were able to semi-automatically synthesize a set of translation rules that they used to compile `ECMAScript` programs to their own instruction set, called JISET. The project comes with a JISET execution engine, which the authors used to test the extracted implementation of `ECMAScript` against Test262, passing 18,064 tests out of 28,952 applicable tests. However, JISET comes with no mechanism for generating an English description of the standard from their JISET implementation, as their goals are exactly the opposite of ours: while we want to obtain the text of the standard from the code of `ECMARef5`, the authors of JISET want to obtain their reference implementation from the text of the standard.

**Comparison with ECMA-SL** Despite the large number of existing implementations of the `ECMAScript` standard, we believe that `ECMA-SL` makes two distinct contributions:

- `ECMARef5` is the most complete academic reference implementation of the `ECMAScript` 5 standard. More concretely, `ECMARef5` passes 12,026 tests out of 12,068 applicable tests, while JSCert [12] passes 1,796 tests, KJS [14] passes 2,782 tests, S5 passes 8,157 tests, and JS-2-JSIL passes 8,797 tests. While it is true that JISET passes 18,064 tests, one has to take into account that JISET targets the tenth version of the `ECMAScript` standard and therefore has a considerably larger pool of available tests (passing only 62,4% of all the applicable tests). More importantly, neither JISET nor any of the existing academic reference implementations have support for all of the `ECMAScript` 5 built-in objects, which we do. Namely, `ECMARef5` is the first academic project that supports the RegExp and the JSON built-in objects, albeit developed in the context of another parallel MSc thesis[1].
- `ECMARef5` is the first `ECMAScript` reference interpreter from which one can obtain an HTML version of the standard, closely matching the official text. We believe that the ideas of this project can be leveraged to improve the current `ECMAScript` standardisation process.

[1] As this thesis is still in progress, we cannot cite it

# Chapter 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 [40].

This chapter presents both the ECMA-SL (4.1) and the Core ECMA-SL (4.2) intermediate languages, also describing the compilation process from ECMA-SL to Core ECMA-SL. We omit the description of the Core ECMA-SL interpreter as it was implemented in the context of another MSc thesis [3].

## 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. Importantly, ECMA-SL does not include the implicit behaviors of the ECMAScript language, such as implicit type coercions and prototype-based inheritance.

An ECMA-SL program is simply a set of top-level functions with a designated entry-point function called main. The complete grammar of ECMA-SL is shown in Figures 4.1 and 4.2, respectively presenting the grammar for ECMA-SL expressions and ECMA-SL statements. It includes three main syntactic categories: (1) values $v \in \mathcal{V}$, (2) expressions $e \in \mathcal{E}$, and (3) statements $st \in \mathcal{S}tmts$. ECMA-SL values include integers, floats, booleans, strings, types, object locations, symbols, the special values null and void, as well as value lists and tuples. We use $(v_1, ..., v_n)$ to denote the n-tuple consisting of values $v_1 - v_n$ and $[v_1, ..., v_n]$ to denote the list containing the elements $v_1 - v_n$. Lists differ from tuples in that they can be extended and shrunk during execution, while tuples always keep the same number of elements.
ECMA-SL expressions can be divided into the following sub-groups:

1. *Simple expressions* consisting of values, local variables ⟨var⟩, and global variables | ⟨var⟩ |;

2. *Operator expressions* consisting of the application of unary, binary, and n-ary operators;

⟨*expr*⟩ ::= ⟨*simple-expr*⟩ | ⟨*operator-expr*⟩ | ⟨*object-expr*⟩ | ⟨*call-expr*⟩ ⟨*simple-expr*⟩ ::= ⟨*value*⟩
| ⟨*var*⟩
| |⟨*var*⟩|

⟨*operator-expr*⟩ ::= ⊖ ⟨*expr*⟩
| ⟨*expr*⟩ ⊕ ⟨*expr*⟩
| ⊗( ⟨*expr*⟩, ..., ⟨*expr*⟩ )

⟨*object-expr*⟩ ::= '{' ⟨*prop*⟩ ':' ⟨*expr*⟩, ..., ⟨*prop*⟩ ':' ⟨*expr*⟩ '}'
| ⟨*expr*⟩ '.' ⟨*prop*⟩
| ⟨*expr*⟩ '[' ⟨*expr*⟩ ']'
| ⟨*expr*⟩ '`in`' ⟨*expr*⟩
| '`fields`' ⟨*expr*⟩

⟨*call-expr*⟩ ::= ⟨*expr*⟩ '(' ⟨*expr*⟩, ..., ⟨*expr*⟩ ')'
| ⟨*expr*⟩ '(' ⟨*expr*⟩, ..., ⟨*expr*⟩ '`) catch `' ⟨*f-name*⟩
| '`extern`' ⟨*expr*⟩ '(' ⟨*expr*⟩, ..., ⟨*expr*⟩ ')'

Figure 4.1: *Syntax of ECMA-SL expressions*. The non-terminals ⟨prop⟩, ⟨var⟩, and ⟨f-name⟩ respectively range over property names, variable names, and function names.

3. *Object expressions* consisting of the object literal expression and static and dynamic property lookup expressions;

4. *Call expressions* consisting of simple function calls, function calls with a catch clause, and external function calls.

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`, as in `ECMAScript`, 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]`)[1].

- *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. Note that, in contrast to normal `ECMA-SL` functions, the `parseJS` function is directly implemented in OCaml, effectively stepping out of the `ECMA-SL` semantics.

[1] Note that `o.foo` is equivalent to `o["foo"]`.

⟨*stmt*⟩ ::= ⟨*simple-stmt*⟩ | ⟨*object-stmt*⟩ | ⟨*conditional-stmt*⟩ | ⟨*loop-stmt*⟩ | ⟨*ret-error-stmt*⟩

⟨*simple-stmt*⟩ ::= ⟨*var*⟩ '`:=`' ⟨*expr*⟩
| '`|`'⟨*var*⟩'`| :=`' ⟨*expr*⟩
| '`{`' ⟨*stmt*⟩ '`;`' ... '`;`' ⟨*stmt*⟩ '`}`'
| '`print`' ⟨*expr*⟩
| '`skip`'

⟨*object-stmt*⟩ ::= ⟨*expr*⟩ '`.`' ⟨*var*⟩ '`:=`' ⟨*expr*⟩
| ⟨*expr*⟩ '`[`' ⟨*expr*⟩ '`]`' '`:=`' ⟨*expr*⟩
| '`delete`' ⟨*expr*⟩ '`[`' ⟨*expr*⟩ '`]`'

⟨*conditional-stmt*⟩ ::= '`switch (`' ⟨*expr*⟩ '`) { case`' ⟨*expr*⟩ '`:`' ⟨*stmt*⟩ ... '`case`' ⟨*expr*⟩ '`:`' ⟨*stmt*⟩ '`}`'
| '`if (`' ⟨*expr*⟩ '`) {`' ⟨*stmt*⟩ '`} elif (`' ⟨*expr*⟩ '`) {`' ⟨*stmt*⟩ '` }`' ... '`else {`' ⟨*stmt*⟩ '`}`'
| '`match`' ⟨*expr*⟩ '`with |`' ⟨*pattern*⟩ '` -> `' ⟨*stmt*⟩ '`|`' ... '`| default -> `' ⟨*stmt*⟩

⟨*pattern*⟩ ::= '`{`' ⟨*pattern-pair*⟩, ..., ⟨*pattern-pair*⟩ '`}`'

⟨*pattern-pair*⟩ ::= ⟨*prop*⟩ '`:`' ⟨*value*⟩
| ⟨*prop*⟩ '`:`' ⟨*var*⟩
| ⟨*prop*⟩ '`: None`'

⟨*loop-stmt*⟩ ::= '`while (`' ⟨*expr*⟩ '`) {`' ⟨*stmt*⟩ '`}`'
| '`repeat {`' ⟨*stmt*⟩ '`}`'
| '`repeat {`' ⟨*stmt*⟩ '`} until`' ⟨*expr*⟩
| '`foreach (`' ⟨*var*⟩ '`:`' ⟨*expr*⟩ '`) {`' ⟨*stmt*⟩ '`}`'

⟨*ret-error-stmt*⟩ ::== '`return `' ⟨*expr*⟩
| '`throw `' ⟨*expr*⟩
| '`fail `' ⟨*expr*⟩

Figure 4.2: *Syntax of ECMA-SL statements*. The non-terminals ⟨prop⟩, ⟨var⟩, and ⟨f-name⟩ respectively range over property names, variable names, and function names.

`ECMA-SL` statements can be divided into the following sub-groups:

1. *Simple statements* include the block statement, the skip statement, the print statement, and the global and local variable assignment statements.

2. *Object statements* include the static and the dynamic property assignment statements and the property delete statement. Note that, as for the case of the property lookup expression, the dynamic property assignment differs from the static property assignment in that the name of the property being assigned must be dynamically computed.

3. *Conditional statements* include the standard if-then-else statement with an arbitrary number of else-if clauses, the switch statement, and the match statement. While the if-then-else and the switch statements have their usual semantics, the match statement requires further consideration. The idea of the match statement is to allow the programmer to match an argument object against multiple object patterns, specifying for each pattern an associated statement clause (analogously to the switch statement). Currently, we support a simple object-pattern language that allows programmers to specify which properties must be present in the argument object and which must not. For instance, consider the following pattern `{ foo: x, bar: "banana", baz: None }`. This pattern specifies that the argument object must have a property `foo` with an arbitrary value and a property `bar` with the value `"banana"`, and must not have a property `baz`; it may, however, have any other properties. Note that the statement clause associated with this pattern may contain the variable `x`, in which case, it binds the value of the property `foo`.

⟨*expr*⟩ ::= ⟨*simple-expr*⟩ | ⟨*operator-expr*⟩

⟨*simple-expr*⟩ ::= ⟨*value*⟩
  | ⟨*var*⟩

⟨*operator-expr*⟩ ::= $\ominus$ ⟨*expr*⟩
  | ⟨*expr*⟩ $\oplus$ ⟨*expr*⟩
  | $\otimes$( ⟨*expr*⟩, ..., ⟨*expr*⟩ )

⟨*stmt*⟩ ::= ⟨*simple-stmt*⟩ | ⟨*object-stmt*⟩ | ⟨*cf-stmt*⟩

⟨*simple-stmt*⟩ ::= ⟨*var*⟩ ‘`:=`’ ⟨*expr*⟩
  | ‘`print`’ ⟨*expr*⟩
  | ‘`skip`’
  | ‘`{`’ ⟨*stmt*⟩ ‘`;`’ ... ‘`;`’ ⟨*stmt*⟩ ‘`}`’

⟨*object-stmt*⟩ ::= ⟨*var*⟩ ‘`:= {}`’
  | ⟨*var*⟩ ‘`:=`’ ⟨*expr*⟩ ‘`[`’ ⟨*expr*⟩ ‘`]`’
  | ⟨*expr*⟩ ‘`[`’ ⟨*expr*⟩ ‘`]`’ ‘`:=`’ ⟨*expr*⟩
  | ‘`delete`’ ⟨*expr*⟩ ‘`[`’ ⟨*expr*⟩ ‘`]`’
  | ⟨*var*⟩ ‘`:=`’ ⟨*expr*⟩ ‘`in`’ ⟨*expr*⟩
  | ⟨*var*⟩ ‘`:= fields`’ ⟨*expr*⟩

⟨*cf-stmt*⟩ ::= ‘`if (`’ ⟨*expr*⟩ ‘`) {`’ ⟨*stmt*⟩ ‘`} else {`’ ⟨*stmt*⟩ ‘`}`’
  | ‘`while (`’ ⟨*expr*⟩ ‘`) {`’ ⟨*stmt*⟩ ‘`}`’
  | ⟨*var*⟩ ‘`:=`’ ⟨*expr*⟩ ‘`(`’ ⟨*expr*⟩, ..., ⟨*expr*⟩ ‘`)`’
  | ‘`return `’ ⟨*expr*⟩
  | ‘`fail `’ ⟨*expr*⟩

Figure 4.3: *Syntax of the Core ECMA-SL language*. The non-terminal ⟨var⟩ ranges over variable names.

4. *Loop statements* include the standard while statement, the bounded and unbounded repeat statements, and the foreach statement. The foreach statement is used to iterate over the elements of a given list. For instance, the foreach statement `foreach(item : lst) { s }` iterates over the items of the list `lst`, executing the body of the foreach statement, `s`, for each element of the list. Note that the variable `item` binds the current list item inside the body of the foreach statement.

5. *Return and error statements* include the return, throw, and fail statements. The return statement has its usual semantics; the throw statement transfers the control to the closest error-handling function in the callstack; and the fail statement simply terminates execution in error mode, as the `exit(1)` command in C-like languages.

## 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. Note that developing an interpreter for `Core ECMA-SL` is substantially simpler than doing so for the entire `ECMA-SL` language. In fact, such an interpreter was developed as part of a parallel MSc thesis [3], focused on the design of a dynamic information flow analysis for `ECMA-SL`. In this section, we describe the `Core ECMA-SL` language and the compiler from `ECMA-SL` to `Core ECMA-SL`.

The syntax of `Core ECMA-SL` is given in Figure 4.3. `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⟩ ']'. Analogously, while ECMA-SL contains a function call expression of the form ⟨expr⟩ '(' ⟨expr⟩, ..., ⟨expr⟩ ')', Core ECMA-SL contains a dedicated function call assignment of the form ⟨var⟩ := ⟨expr⟩ '(' ⟨expr⟩, ..., ⟨expr⟩ ')'. The same holds for the membership-check, empty object literal, and property collection expressions, which were all promoted to dedicated assignment statements.

The ECMA-SL to Core ECMA-SL compiler was implemented in the OCaml programming language. It 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. Below, we describe this compilation process by appealing to five illustrative examples, each focusing on one of the five aspects stated above.

**Side-effect-free heap-independent expressions** As stated above, Core ECMA-SL expressions neither have side-effects nor interact with the heap. To this end, the ECMA-SL expressions that include these features were promoted to dedicated Core ECMA-SL assignment statements. Hence, our Core ECMA-SL expression compiler, compile_expr, compiles the given expression to a pair consisting of a statement that captures the effectful behavior of the given expression and a side-effect-free expression that holds the value of the original expression. Consider the example below.

```
1 z := o1[x] + o2[y]
```

```
1 __v0 := o1[x];
2 __v1 := o2[y];
3 z := __v0 + __v1
```

Figure 4.4: Example of effect free expressions in ECMA-SL (left) together with compiled to Core ECMA-SL version (right).

On the left, we show an ECMA-SL statement and on the right its Core ECMA-SL counter-part. We can see that the original assignment includes the expression o1[x] + o2[y] that interacts with the heap. The application of compile_expr to this expression yields the statements __v0 := o1[x]; __v1 := o2[y], which capture the heap-manipulating logic of the expression, and the expression __v0 + __v1, which holds the value of the original expression after the execution of its effectful statements. Using the output of compile_expr, the function compile_stmt constructs the statement that is shown on the right in the expected way.

**Single loop statement** As Core ECMA-SL only includes the standard while statement, the bounded and unbounded repeat statements, as well as the foreach statement, are compiled to equivalent while statements. The compilation of these three loop statements is straightforward. In the following, we give an example of the compilation of the foreach statement.

```
1 foreach(p : lst) {
2   print(p)
3 }
```

```
1 i := 0;
2 list_length := l_len(lst);
3 while (list_length > i) {
4   p := l_nth(lst, i);
5   print p;
6   i := i + 1
7 }
```

Figure 4.5: Example of a foreach statement in ECMA-SL (left) together with compiled to Core ECMA-SL version (right).

On the left, we show an ECMA-SL statement and on the right its compilation to Core ECMA-SL. Recall that the foreach statement is used to iterate over the elements of a list. Hence, in the compiled code, we simply generate an iterating variable i that keeps track of the current index of the list. At each iteration, we start by obtaining the element of the list at index i and assigning it to p. Then, we execute the body of the foreach statement. And, finally, we increment the current list index. The loop terminates when the current index coincides with the value of the list length.

**Single conditional statement** Match statements and switch statements are compiled to sequences of if-then-else statements. In contrast to the compilation of switch statements, which is straightforward, the compilation of match statements is substantially more involved. In particular, we designed an auxiliary compiler that given a simple object pattern, returns the sequence of checks that must hold for the pattern clause to be executed. Consider the example given in Figure 4.6. On the left, we present the

```
1 match obj with
2 | { foo: 42, baz: None } -> {
3   print "banana"
4 }
5 | default -> {
6   print "apple"
7 }
```

```
1   __v0 := "foo" in_obj obj;
2   __v1 := obj["foo"];
3   __v2 := __v1 = 42;
4   __v3 := "baz" in_obj obj;
5   __v3 := !(__v3);
6   if (__v0 && __v2 && __v3) {
7     print "banana"
8   } else {
9     print "apple"
10 }
```

Figure 4.6: Example of a match statement in ECMA-SL (left) together with compiled to Core ECMA-SL version (right).

ECMA-SL match statement and on the right its compilation to Core ECMA-SL. The pattern { foo: 42, baz: None } matches objects that have the property foo with the value 42 and that do not have the property baz. This pattern is compiled to the sequence of statements given in lines 1-5 on the right-hand side of Figure 4.6 as well as the expression __v0 && __v2 && __v3. Intuitively, __v0 is true if the object obj has the property foo, __v2 is true if the property foo has value 42, and __v3 is true if the object obj does not have the property baz. Hence, if the expression __v0 && __v2 && __v3 is true the compiled program executes the statement clause associated with the pattern; otherwise, it executes the default clause.

**No support for global variables** The ECMA-SL language only contains top-level functions. This means that functions cannot be nested inside each other and each function executes in its own scope. However, ECMA-SL does allow programmers to make use of global variables, which are shared by all functions. In contrast, Core ECMA-SL does not support global variables. To account for them, the compiler from ECMA-SL to Core ECMA-SL adds the extra parameter `___globals` to all function definitions, with the goal of simulating the global variable scope. The compilation of function definitions must fit together with the compilation of call expressions. To this end, call expressions are compiled to equivalent call statements that include the additional argument `___globals`. Consider the example below.

```
function f() {
  |foo| := 3;
  bar := |foo| + 2;
  ...
}
```

```
function f(___globals) {
  ___globals["foo"] := 3;
  __v0 := ___globals["foo"];
  bar := __v0 + 2;
  ...
}
```

Figure 4.7: Example of accessing global variables in ECMA-SL (left) together with compiled to Core ECMA-SL version (right).

The program on the left makes use of a global variable `|foo|`, both updating and reading its value. The compiled program updates the value of `foo` and reads its value directly on/from the object bound to the parameter `___globals`.

**No support for errors** Core ECMA-SL does not support the throw statement and the call expression with catch clause. To account for this, the compiler from ECMA-SL to Core ECMA-SL instruments the return of each function with an additional Boolean flag indicating whether or not the function has thrown an error. More concretely, the statement `throw e` is compiled to the statement `return (true, e)`, where the Boolean `true` indicates that the function threw an error. Following the same logic, the statement `return e` is compiled to the statement `return (false, e)`, indicating that the function returned normally. Naturally, the compilation of call expressions must check if the called function threw an error and proceed accordingly; there are three scenarios to consider:

- the execution of the called function threw an error and the programmer specified an error handler;
- the execution of the called function threw an error and no error handler was specified;
- the execution of the called function returned normally.

```
function f(y) {
   x := g(y) catch h
   ...
}
function g(x) {
    if (x > 0) {
       ...
       return x + 1
    } else {
       throw 0
    }
}
```

```
function f(___globals, y) {
  __v0 := g(___globals, y);
  if (fst(__v0)) {
    __v0 := h(___globals, snd(__v0));
    if (fst(__v0)) {
      return __v0
    } else {
      __v0 := snd(__v0)
    }
  } else {
    __v0 := snd(__v0)
  };
  ...
}
```

Figure 4.8: Example of a function call with guard in ECMA-SL (left) together with compiled to Core ECMA-SL version (right).

Figure 4.8 shows the compilation of a function call with catch clause. The compiled code first checks if the first element of the pair returned by `g` is `true`. If it is, the compiled code calls the error handler `h` giving it the error value as an argument (recall that the error value is the second element of the pair). If it is not, the compiled code simply extracts the value returned by `g` from the pair and assigns it to `__v0`. Note that, the handler `h` may itself throw an error. In such cases, the compiled function `f` returns immediately.

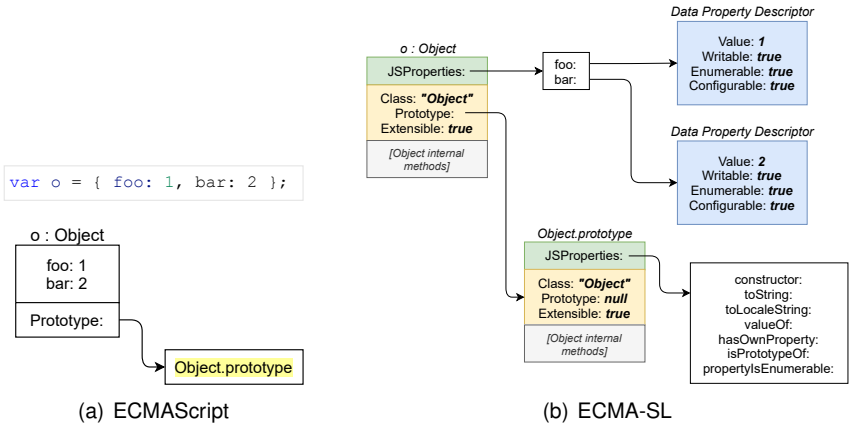
# Chapter 5

# 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 explaining the internal representations that we have used to model the different types of artifacts used in the ECMAScript standard. More specifically, we discuss our internal representations of: (1) ECMAScript objects; (2) property descriptors; and (3) function objects.




(a) ECMAScript (b) ECMA-SL

Figure 5.1: ECMAScript and ECMA-SL objects representations.

**ECMAScript objects** As discussed in Subsection 2.1.2, 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 5.1 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. For clarity, we refer to the main object as `o` and leave the auxiliary object unnamed, as it is always accessed through the main one. In the following, we represent auxiliary objects as white boxes and main objects as three-colored boxes: a green segment pointing to the auxiliary object; a yellow segment storing its internal properties; and a grey segment storing the internal methods that can be applied to that object.

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.

Note that the alternative decomposition, which stores the named properties in the main object and the internal properties in the auxiliary object, does not work as it leads to the same property-clash problem. To understand this issue, let us assume that we implement the alternative decomposition and connect the main object to the auxiliary object via a dedicated property `internalProps`. How can we guarantee that there are no clashes between named properties and the property `internalProps`? We get back to stage zero.

```
object ECMAScriptObject {
   constructor(string class, boolean extensible, ECMAScriptObject prototype);
   attribute string Class;
   attribute boolean Extensible;
   attribute ECMAScriptObject? Prototype;
   attribute dictionary<string, any> JSProperties;

   private any getJSProperty(string propertyName);
   private void setJSProperty(string propertyName, any propertyValue);
   private any getInternalProperty(string internalPropertyName);
   private void setInternalProperty(string internalPropertyName, any internalPropertyValue);

   [section 8.12]
   internal [[GetOwnProperty]](string P);
   internal [[GetProperty]](string P); ...
}
```

Listing 5.1: Interface of ECMAScript objects in ECMA-SL.

Finally, Listing 5.1 describes the internal structure of `ECMAScript` objects as they are implemented in `ECMA-SL` using *IDL*, a standard interface description language. We can see that every `ECMAScript` object defines the properties `Class`, `Extensible`, and `Prototype`, respectively storing a string, a boolean, and an `ECMAScript` object, as well as the property `JSProperties`, storing a key-value dictionary with the named properties of that object. Every `ECMAScript` object also contains properties for keeping the internal methods that can be applied to it. For instance, the internal methods `GetProperty` and `GetOwnProperty`. Besides these internal properties and internal methods, we also define some auxiliary methods useful for getting or setting either internal properties or named properties. For instance, `getJSProperty` and `setInternalProperty`.

**Property descriptors** As previously stated, `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`. As discussed in Subsection 2.1.3, the `ECMAScript` standard mandates that named properties be represented by *property descriptors*. Each property descriptor

is a record containing specific attributes representing both the property value and meta-information about the property. 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 5.1. 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; for the foo property Value: 1 and for the bar property Value: 2. Additionally, each descriptor has the three meta-properties that fully populate the corresponding data property descriptor: Writable, Enumerable, and Configurable. All these meta-properties have value true since it is their default value.

```
object PropertyDescriptor {
   attribute boolean Enumerable;
   attribute boolean Configurable;
}

object DataPropertyDescriptor : PropertyDescriptor {
   constructor(any value, boolean writable, boolean enumerable, boolean configurable);
   attribute boolean Writable;
   attribute any Value;
}

object AccessorPropertyDescriptor : PropertyDescriptor {
   constructor(getter get, setter set, boolean enumerable, boolean configurable);
   getter any Get?();
   setter void Set?(any value);
}
```

Listing 5.2: Interface of Property Descriptors in ECMA-SL.

Finally, Listing 5.2 describes in IDL the internal structure of ECMAScript property descriptors as they are implemented in ECMA-SL. We define a super-object PropertyDescriptor containing the two boolean properties that both data property descriptors and accessor property descriptors must have, Enumerable and Configurable. Then, we can see the concrete definition of the two types of property descriptors. Each definition only contains the properties that correspond to the specific type of property descriptor. Note that we indicate that the two specific properties of accessor property descriptors may be absent. This illustrates exactly what is expressed in Table 6 of the ECMAScript standard. Besides these properties, we also define the constructors that help with the creation of property descriptors in ECMA-SL.

**Function objects** Function objects are a special type of ECMAScript object used to represent functions. A function object stores meta-information regarding the corresponding function in dedicated internal properties; for instance, it stores its list of parameters in the property [[FormalParameters]] and the internal representation of its body in the property [[Code]]. Furthermore, programmers can add named properties to function objects.

In ECMA-SL, we represent function objects as a special type of ECMAScriptObject with six additional internal properties: Call, Code, Construct, FormalParameters, HasInstance, and Scope. The properties Call, Construct, and HasInstance store the identifiers of the ECMA-SL functions used to interpret function calls, constructor calls, and the instanceOf expression. The remaining properties store meta-information about the function; namely, its list of parameters (FormalParameters), its body (Code), and the scope in which it was created (Scope).

Listing 5.3 describes in IDL the internal structure of ECMAScript Function Objects as they are implemented in ECMA-SL.

```
object FunctionObject : ECMAScriptObject {
   attribute string Call;
   attribute oneOf<string, object> Code;
   attribute string Construct?;
   attribute list<any> FormalParameters;
   attribute string HasInstance;
   attribute LexicalEnvironment Scope;
}
```

Listing 5.3: Interface of Function objects in ECMA-SL.

The property Code of function objects is worth discussing in detail as it can store values of different types. If a function object represents a function created by the programmer, its Code property stores the corresponding AST; more precisely, it stores the ECMA-SL object corresponding to the root of the AST. If a function object represents a built-in function, its Code property stores the identifier of the ECMA-SL function that implements the corresponding algorithm; for instance, the function object corresponding to String.prototype.split stores the identifier StringPrototypeSplit.

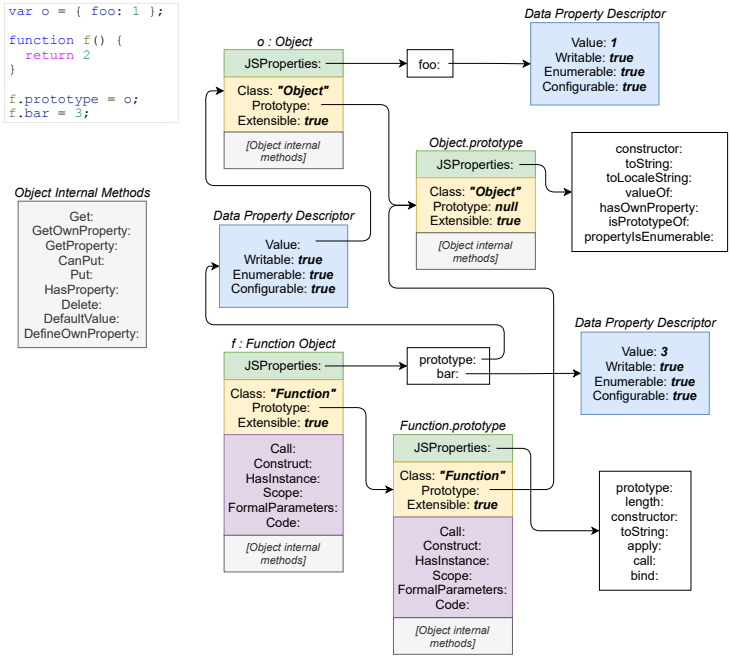



Figure 5.2: Function Objects' internal representation.

Finally, Figure 5.2 shows an ECMAScript code snippet together with the ECMA-SL object graph resulting from its execution. The code snippet creates a function f with prototype o and assigns the value 3 to the property bar of f. In the heap, function objects are represented as normal ECMAScript objects except that they contain a further purple segment with the internal properties that are specific to function objects. We can see that function f has two named properties: bar and prototype. The property bar stores a data property descriptor with value 3. The property prototype stores a data property descriptor whose value points to object o.

## 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 func-

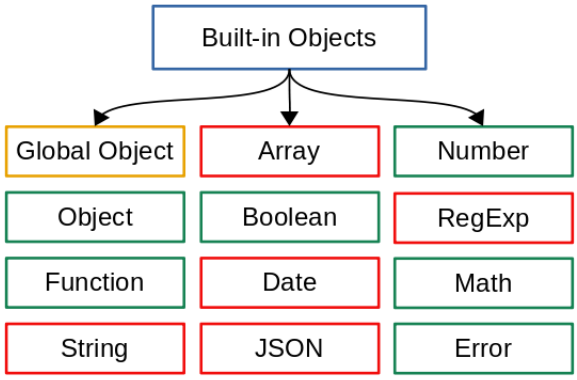



Figure 5.3: Standard Built-in ECMAScript Objects: in green are the ones we implemented; in yellow are the ones we partly implemented; and, in red are the ones implemented as part of other projects.

tions are made available to the programmer via dedicated built-in objects; for instance, most regular expression functions are exposed via the RegExp built-in object. Built-in objects are created in the heap whenever an `ECMAScript` program executes. They include 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 various *Error objects*. We refer to the heap that *only* contains the `ECMAScript` built-in objects as the *initial heap*.

**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; four of its functions are still ongoing (`decodeURI`, `decodeURIComponent`, `encodeURI`, and `encodeURIComponent`); we expect to have these functions implemented by the time this thesis is presented. However, not all built-in objects were implemented in the context of this thesis. Figure 5.3 illustrates our relative contribution to the implementation effort, showing in green the built-in objects that we implemented and in red those that we did not.

**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. For instance, the value of the internal property `[[Prototype]]` of the function constructor is the function prototype object and the value of the named property `constructor` of the function prototype object is the function constructor. This circular dependency is illustrated in blue in Figure 5.4. Circular dependencies may involve more than two built-in objects. In Figure 5.4, we show in red a circular dependency involving the objects: Object Constructor, Function Prototype, and Object Prototype.

In order to cope with the circular dependencies discussed above, we have to postpone the creation of certain properties when initialising their corresponding objects. As an example, let us consider the mutual dependency between the Function Constructor and the Function Prototype shown in blue in Figure 5.4. In order to create these two objects, we proceed as follows:

- We first create the Function Prototype object without initialising its named property `constructor` (note that, as a rule, we create prototype objects first and only then their corresponding constructors);
- We create the Function Constructor object, setting its internal property `Prototype` to the Function Prototype object created in step 1;

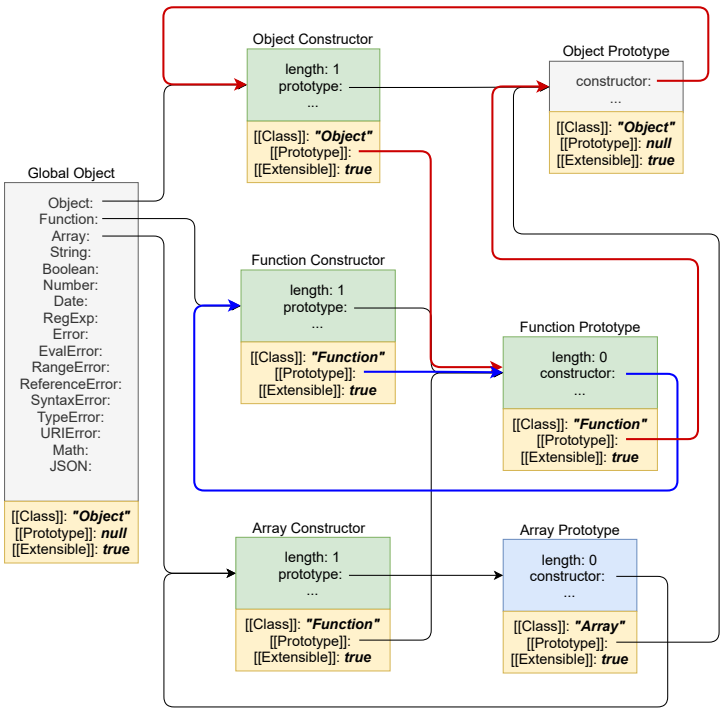



Figure 5.4: Global Object's built-in objects circular dependencies.

- We set the `constructor` named property of the Function Prototype object to the Function Constructor created in step 2.

The initialisation process is managed by the function `initGlobalObject`, which is used to create the entire initial heap. Note that inter-dependencies become more difficult to manage as we consider longer dependency chains. It is the job of the `initGlobalObject` to create the various built-in objects in the appropriate order, returning a pointer to the Global Object through which all of the built-in objects can be accessed. In Figure 5.5, we give the fragment of the function `initGlobalObject` that creates the Function Constructor and the Function Prototype objects.

The first three statements of `initGlobalObject` create and initialise the Global Object, the Function Prototype object, and the Function Constructor object, without initialising the properties that would generate a circular dependency between them; for instance, it does not initialise the `constructor` named property of the Function Prototype object. After creating the Function Prototype and the Function Constructor objects, we can now establish the circular dependency. To this end, in line 16, we initialise the `constructor` named property of the Function Prototype object, setting it to a data property descriptor whose `Value` attribute is the Function Constructor object. The remaining three attributes are set to `false`, since Subsection 15.3.3.1 of the `ECMAScript` standard mandates that the property `constructor` of the Function Prototype object be non-writable, non-enumerable, and non-configurable. Finally, in line 22, we use the auxiliary function `createBuiltInProperty` to assign the Function Constructor to the property `Function` of the Global Object.

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. In Figure 5.5, on the right, we show the serialisation of the fragment of the initial heap constructed on the left, which is represented diagrammatically in Figure 5.6.

The initial heap that is the result of the execution of the `ECMA-SL` example detailed above is illustrated on the right-side of the Figure 5.5.

```
function initGlobalObject() {

 /* create $l_glob and $l_glob_props */
 global := NewECMAScriptObjectFull(
   'null, "Object", true);

 /* create $l_fproto, $l_fproto_props,
    $l_dd_apply, and $l_apply */
 FunctionPrototype :=
   initFunctionPrototype(global);

 /* create $l_fconstr, $l_fconstr_props and
    $l_dd_length */
 FunctionConstructor :=
   initFunctionConstructor(global);

 /* create $l_dd_fproto and add the property
     "prototype" to $l_fconstr_props */
 createBuiltInPropertyWithFullDescriptor
   (FunctionConstructor, "prototype",
    FunctionPrototype, false, false,
    false);

 /* create $l_dd_fconstr and add the
    property "Function" to $l_glob_props */
 createBuiltInProperty(global, "Function",
    FunctionConstructor);

 ...

 return global
}
```

```
{
  "$l_glob": {
    "JSProperties": "$l_glob_props", ...
  },
  "$l_glob_props": {
    "Function": "$l_dd_fconstr"
  },
  "$l_fproto": {
    "JSProperties": "$l_fproto_props", ...
  },
  "$l_fproto_props": {
    "apply": "$l_dd_apply", ...
  },
  "$l_dd_apply": {
    "Value": "$l_apply", ...
  },
  "$l_apply": {
    "Prototype": "$l_fproto",
    "Code": "FunctionPrototypeApply", ...
  },
  "$l_fconstr": {
    "JSProperties": "$l_fconstr_props",
    "Prototype": "$l_fproto", ...
  },
  "$l_fconstr_props": {
    "prototype": "$l_dd_fproto",
    "length": "$l_dd_length"
  },
  "$l_dd_length": {
    "Value": 1, ...
  },
  "$l_dd_fproto": {
    "Value": "$l_fproto", ...
  },
  "$l_dd_fconstr": {
    "Value": "$l_fconstr", ...
  }
}
```

Figure 5.5: Example of the *GlobalObject* initialisation in ECMA-SL (left) with corresponding Heap JSON serialisation (right).

## 5.3 Line-by-line Closeness

We demonstrate that ECMARef5 follows the ECMAScript standard line-by-line by example. We consider the internal functions [[GetProperty]] and [[GetOwnProperty]], already discussed in Subsection 2.1.12, the new operator, and the while and try statements, presenting for each example both the pseudo-code of the standard and our ECMA-SL implementation. Figures 5.7-(a) and 5.7-(b) show our implementations of GetOwnProperty and GetProperty, while Figures 5.12-(a), 5.12-(b), and 5.12-(c) show our implementations of the new operator, While statement, and Try statement. One can easily see that the given ECMA-SL implementations precisely coincide with the text of the standard. In the following, we will briefly explain the examples, highlighting their specific challenges from the point-of-view of the design of ECMA-SL.

**Internal Functions** 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. For instance, if statements are mapped to ECMA-SL if statements, let assignments are mapped to ECMA-SL variable assignments, return statements are mapped to ECMA-SL return statements, and property updates are mapped to ECMA-SL property updates. However, in order to streamline the interaction with our internal representation of ECMAScript objects, we make use of a range of auxiliary functions, such as: (1) getJSProperty(O, P) for obtaining the property descriptor associated with P in O; (2) NewPropertyDescriptor() for creating a new empty property descriptor; (3) IsDataPropertyDescriptor(X) for checking if X is a data property descriptor; and (4)

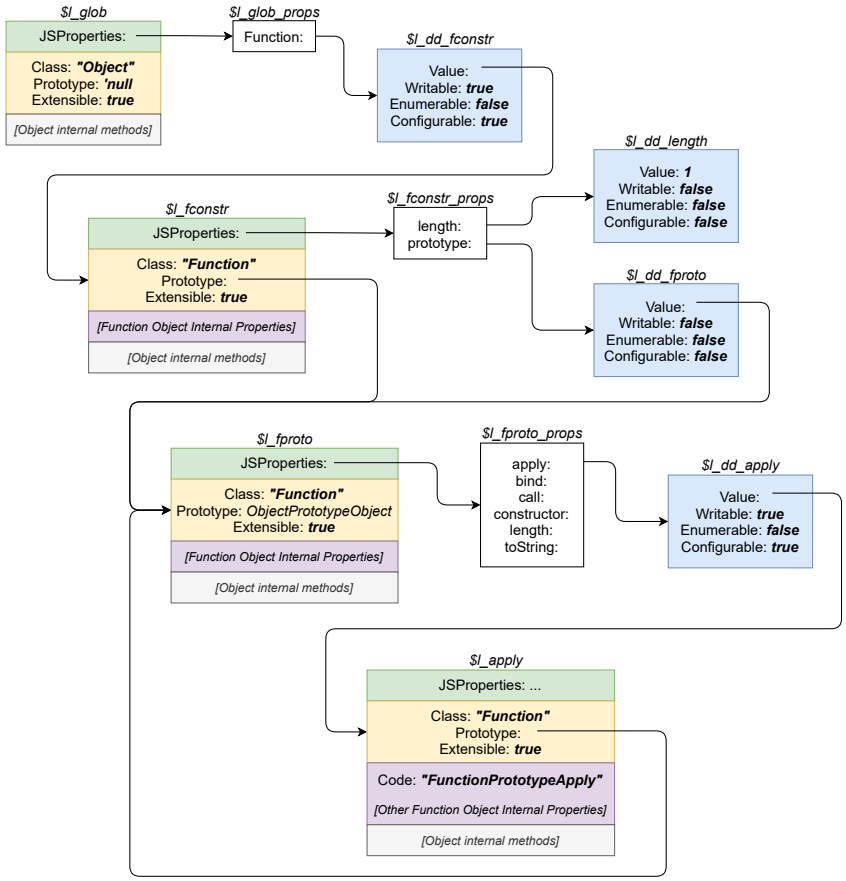



Figure 5.6: Objects representation matching the initialisation of the GlobalObject.

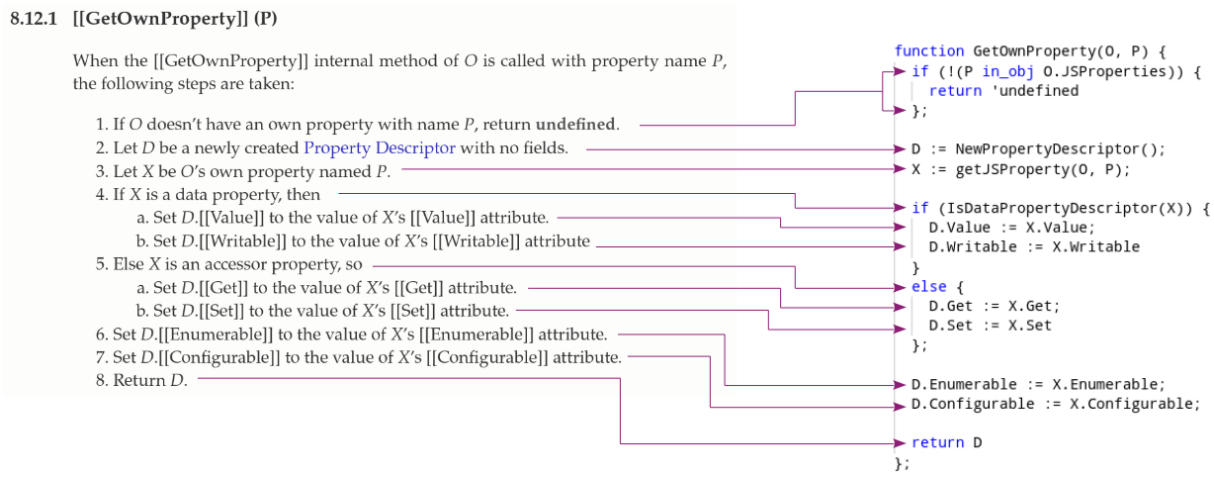
`getInternalProperty(O, P)` for obtaining the internal property `P` of `O`.

It is worth noting the use of dynamic function calls in our `ECMA-SL` implementations of internal functions. For instance, in the code of `[[GetProperty]]`, the expression `{O.GetOwnProperty}(O, P)` will call the function bound to the property `GetOwnProperty` of `O` with parameters `O` and `P`. In contrast to the `ECMAScript` language, `ECMA-SL` does not include the keyword `this`; hence, we have to pass `O` as a parameter in order to be able to refer to it from within the code of `GetOwnProperty`.

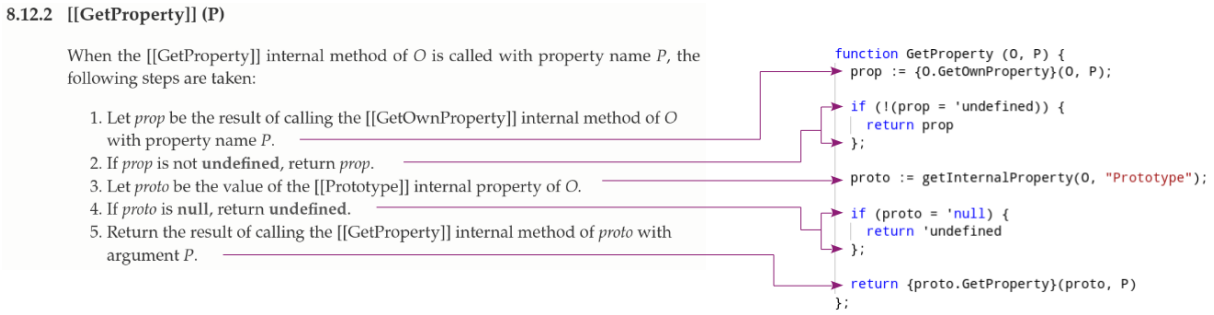
**Expressions and Statements** Implementing the `ECMAScript` expressions and statements in `ECMA-SL` required adding support for pattern matching over `ECMA-SL` objects in the form of a simple match statement. Hence, in the interpretation of statements/expressions, we start by matching the statement/expression given as input against all possible types of statements/expressions. In Figure 5.8, we give a code snippet of the main match statement used for interpreting `ECMAScript` statements.

The different types of `ECMAScript` statements/expressions are represented by `ECMA-SL` objects whose structure is given by the Esprima version of the *ESTree* specification [41], a standardised AST format for representing `ECMAScript` programs. In particular, every statement/expression is modeled as an object with a property `type`, which identifies its type. For instance, while statements have type `"WhileStatement"` and try statements have type `"TryStatement"`.

Consider the example of the try statement. The standard has several productions associated with it, depending on whether or not it contains a catch clause and/or a finally clause. Accordingly, our interpreter has several patterns, each corresponding to one of the productions. For instance, we have a pattern for the case when there is a catch clause but no finally clause, a pattern for the opposite




(a) The specification of the Object internal function `[[GetOwnProperty]]` and the corresponding `ECMA-SL` code.




(b) The specification of the Object internal function `[[GetProperty]]` and the corresponding ECMA-SL code.

Figure 5.7: `GetOwnProperty` and `GetProperty` specifications and corresponding `ECMA-SL` code.

case (finally clause and no catch clause), and a final pattern for when both clauses exist. Note that the absence of a given clause in the try statement is denoted by the `null` value in its corresponding pattern.

## 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 given in Figure 5.9, and an optimised one given in Figure 5.11. 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* [42], a standard-compliant `ECMAScript`

```
function JS_Interpreter_Stmt(stmt, scope) {
  match stmt with
  | { type: "ExpressionStatement", expression: Expression } -> {
    ...
  }
  | { type: "WhileStatement", test: Expression, body: Statement, labelSet: currentLabelSet } ->
    {
    ...
  }
  | { type: "TryStatement", block: Block, handler: Catch, finalizer: null } -> {
    ...
  }
  | { type: "TryStatement", block: Block, handler: null, finalizer: Finally } -> {
    ...
  }
  | { type: "TryStatement", block: Block, handler: Catch, finalizer: Finally } -> {
    ...
  }
  ...
};
```

Figure 5.8: Code snippet of the main match statement used for interpreting ECMAScript statements

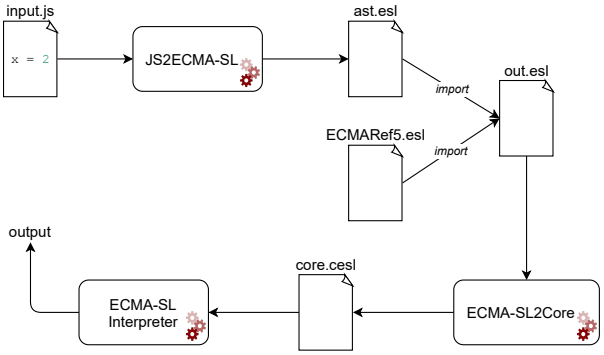
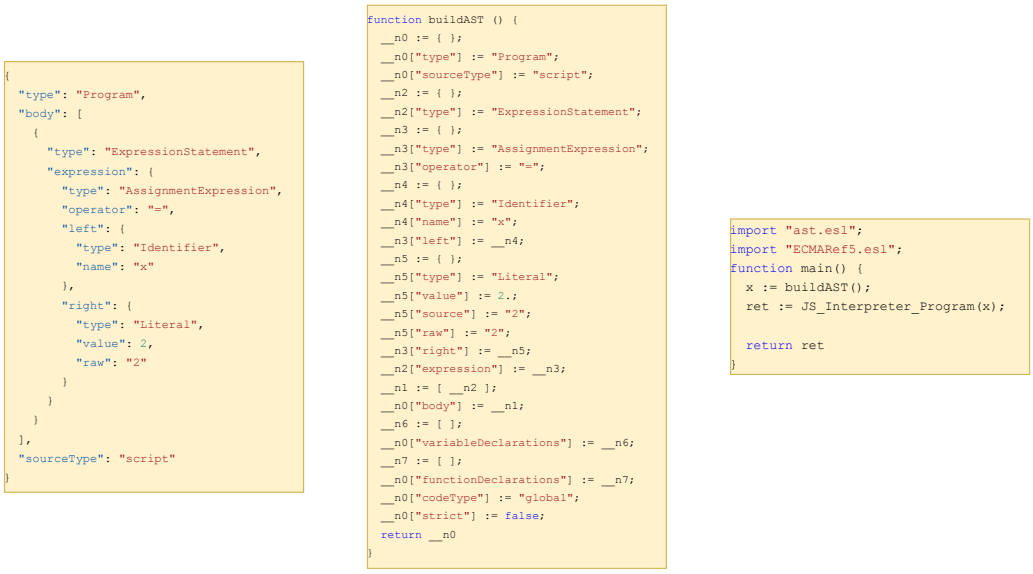



Figure 5.9: ECMAScript file execution pipeline.

parser. 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`, which corresponds to the AST object given in Figure 5.10-(a), is transformed into the ECMA-SL function `buildAST` shown in Figure 5.10-(b). Note that 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, `ECMARef5.esl`, and the generated `buildAST` function, `ast.esl`, and simply calls the interpreter on the result of `buildAST`. This program, which corresponds to the compilation of `input.js` to ECMA-SL, is given in Figure 5.10-(c).

**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, `core.cesl`, is completely self-contained, including all the code of ECMARef5 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



(a) Syntax tree object (b) ECMA-SL function (c) out.esl function

Figure 5.10: Syntax tree object and corresponding generated ECMA-SL function for the ECMAScript program `x = 2`

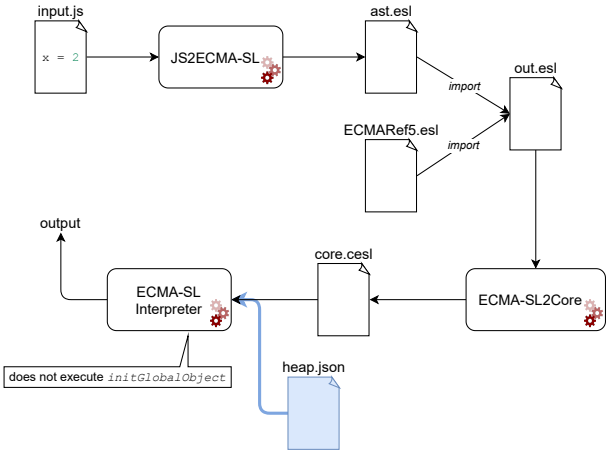
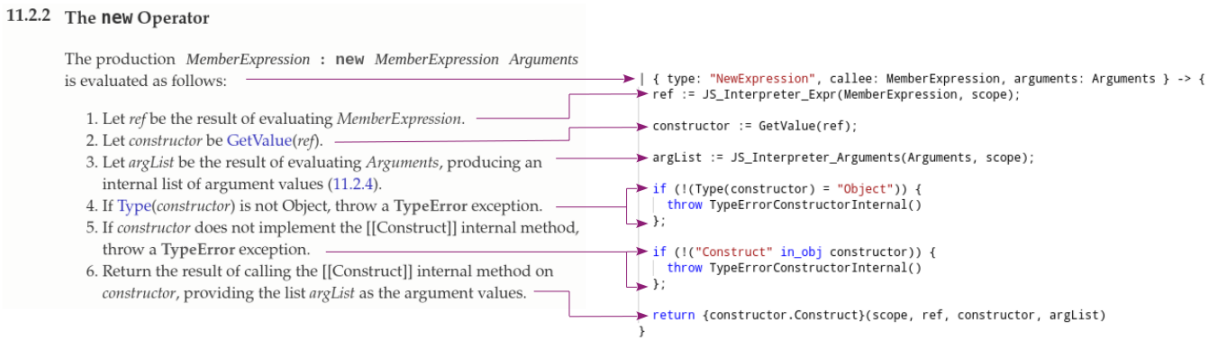


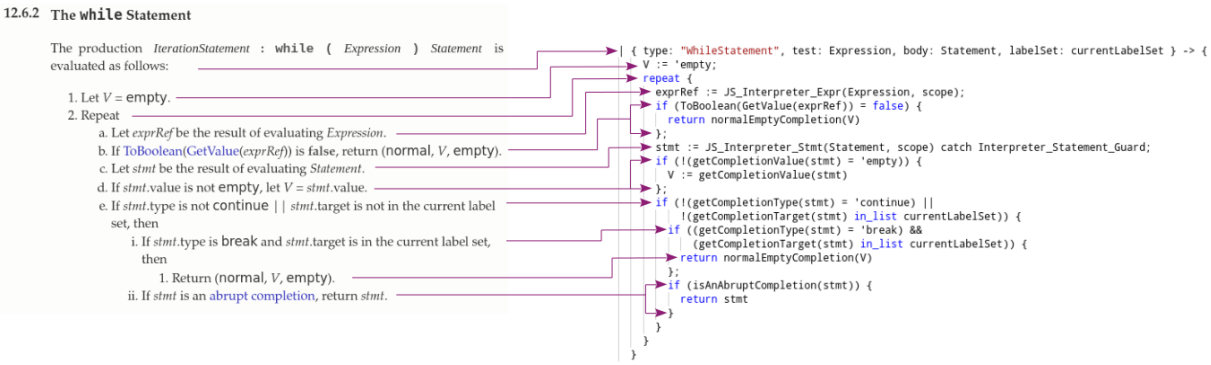

Figure 5.11: ECMAScript file execution pipeline with optimisation.

mode, the interpreter additionally logs the sequence of executed commands for debugging purposes.

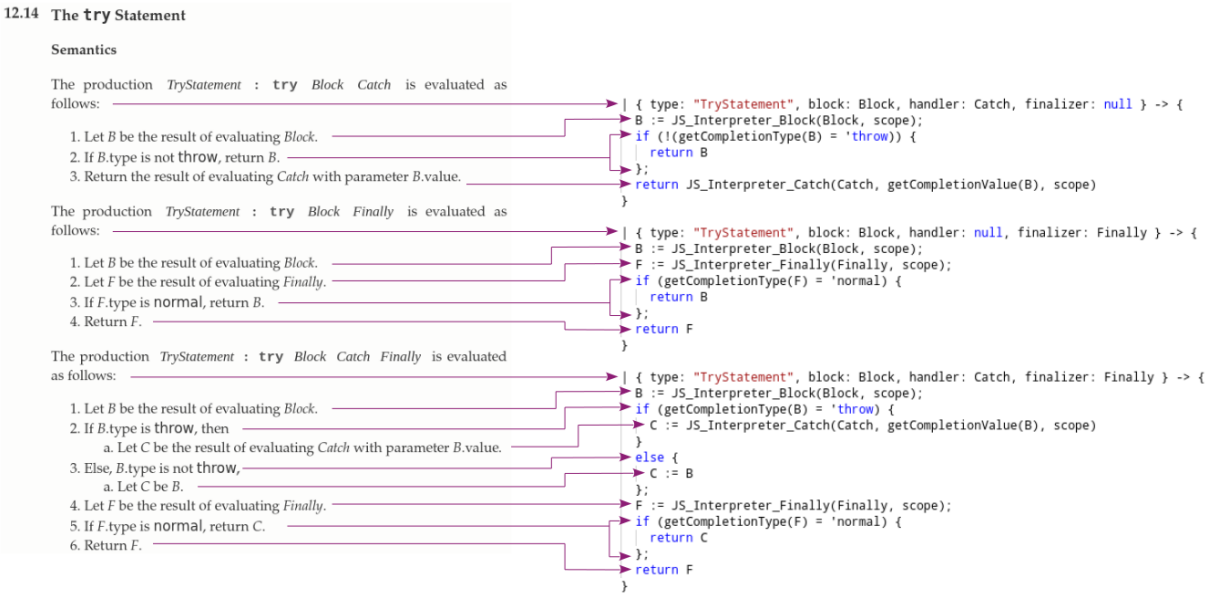
**Optimised Pipeline** When interpreting an `ECMAScript` program, one starts by constructing the initial `ECMAScript` heap, which contains the Global Object as well as all the other built-in objects described in Section 2.1. The simplest way to set up this initial heap is to actually execute the `ECMA-SL` code that constructs its objects. In fact, our non-optimised pipeline simply calls the function `initGlobalObject` which creates the entire initial heap. 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. Figure 5.11 presents the optimised execution pipeline. The optimization consists in not recomputing the initial heap from scratch but instead loading it from a previously generated JSON file. This additional step is performed before start the interpretation of the `Core ECMA-SL` program.




(a) The specification of the `new` expression and the corresponding `ECMA-SL` code.




(b) The specification of the `While` statement and the corresponding `ECMA-SL` code.




(c) The specification of the `Try` statement and the corresponding `ECMA-SL` code.

Figure 5.12: The `new` operator, `While` statement, and `Try` statement specifications and corresponding `ECMA-SL` code.

# Chapter 6

# HTML Generator

ECMARef5 follows the ECMAScript standard line-by-line. This methodology has been proposed in other projects [12; 8; 32] as a means to establish trust in the reference interpreter. However, no prior project has quantified the closeness between the code of the corresponding ECMAScript implementation and the text of the standard. In this chapter, we describe ECMA-SL2English, our tool for generating the text of the standard from the code of ECMARef5, which we use to automatically generate an HTML version of the standard. In the next chapter, we use out-of-the-box text-comparison metrics to measure the closeness between the generated standard and its official version.

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, 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. We decided to go with an implementation that generates the HTML corresponding to the first example, since most of the occurrences in the standard follow that pattern. Also, this happens in various other function calls, not just calls to the internal method [[GetOwnProperty]] or calls to functions that only receive one argument.

Although ECMA-SL is very close to 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 of calling [[GetProperty]]; hence, it cannot be inferred by our reference implementation.

```
6. If IsAccessorDescriptor(desc) is true, then
    a. Let setter be desc.[[Set]] (see 8.10) which cannot be undefined.
    b. Call the [[Call]] internal method of setter providing base as the this value and an argument list containing only W.
7. Else, this is a request to create an own property on the transient object O
    a. If Throw is true, then throw a TypeError exception.
```

```
2. If desc is undefined, then return true.
3. If desc.[[Configurable]] is true, then
    a. Remove the own property with name P from O.
    b. Return true.
4. Else if Throw, then throw a TypeError exception.

```

Figure 6.1: Excerpts of the ECMAScript standard: 8.7.2 PutValue (left); and 8.12.7 [[Delete]] (right).

- 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. Figure 6.1 illustrates six different textual representations for if-else statements. In particular, it shows that sometimes the standard includes a line break after the keyword `then`, while other times it does not; and analogously for the keyword `else`. Furthermore, sometimes an if-else statement has its own separate item, while other times it is represented as part of the item of the construct where it occurs.

In order to allow for greater flexibility when generating the text of a given ECMA-SL construct, we extend ECMA-SL with a set of code generation directives that we use to determine the way each construct is to be turned into text. Furthermore, as we did not want to hard-code in ECMA-SL2English the code generation patterns specific to the ECMAScript standard, we made it parametric on a set of code generation rules that are fed to the tool in JSON format. The support for code generation directives and rules makes ECMA-SL2English a highly flexible tool, which can be readily adapted to the more recent versions of the standard.

We organize this chapter into three sections. In the Section 6.1, we present the detailed description of the HTML structure of the ECMAScript standard. In Section 6.2, we present the main code generation algorithm underpinning the ECMA-SL2English tool. Finally, in Section 6.3, we detail the code generation directives and rules that we use to determine the way each construct is to be turned into text.

## 6.1 HTML Structure of the ECMAScript Standard

The structure of the generated HTML version of the ECMAScript standard must coincide with the structure of its official version. To accomplish this, we start by characterising the official HTML structure of the standard. It is important to note that we only generate the parts of the standard that were included in our reference interpreter. For instance, we do not generate parts of the standard that only consist of textual description of internal data structures.

The ECMAScript standard is composed of five main types of organisational elements: sections, subsections, sub-sub-sections, algorithms, and production rules. These elements are all represented in the HTML version of the standard as `<section>` elements with an attribute `id` storing their unique identifiers. In the following, we describe the HTML structure of ECMAScript algorithms and semantic production rules. We do not describe the HTML structure of syntactic production rules as they are not modelled by our interpreter.

**HTML structure of ECMAScript algorithms** Each algorithm presented in the standard corresponds to an HTML `<section>` element with the following inner elements:

```
<section id="sec-8.12.1">
  <h1>
    <span class="secnum">
      <a href="#sec-8.12.1" title="link to this section">8.12.1</a>
    </span>
    [[GetOwnProperty]] (P)
  </h1>

  <p>When the [[GetOwnProperty]] internal method of <var>O</var> is called ... </p>

  <ol class="proc">
    ...
    <li>If <i>X</i> is a data property, then
      <ol class="block">
        <li>Set <i>D</i>.[[Value]] to the value of <i>X</i>'s [[Value]] attribute.</li>
        <li>Set <i>D</i>.[[Writable]] to the value of <i>X</i>'s [[Writable]] attribute</li>
      </ol>
    </li>
    ...
  </ol>

  <p>However, if <var>O</var> is a String object it has ... </p>
</section>

```

Figure 6.2: A fraction of the HTML code corresponding to the internal function `[[GetOwnProperty]]`. See Figure 5.7 for the corresponding snippet of the standard.

1. an HTML heading, `<h1>`, containing an HTML anchor element, `<a>`, with the corresponding section number and title;
2. an optional HTML paragraph element, `<p>`, containing a textual description of the algorithm;
3. an HTML ordered list element, `<ol>`, containing the pseudo-code of the algorithm, where each step of the algorithm is represented as an HTML list item, `<li>`. Note that, a step of the algorithm may require performing a sequence of sub-steps, in which case, the list item associated with the main step contains an inner HTML ordered list with the sub-steps;
4. a final optional HTML paragraph containing further notes and details about the algorithm.

For instance, the internal function `[[GetOwnProperty]]` corresponds to the HTML code given in Figure 6.2.

Even though most of the algorithms of the standard are described in operational style, a few algorithms are described in declarative style using tables. For instance, the description of the `ToPrimitive` abstract operation, presented in Subsection 9.1 of the standard, contains the table shown in Figure 6.3, which maps each possible type of input to the expected result.

**HTML structure of ECMAScript semantic productions** Some expressions/statements of `ECMAScript` are associated with multiple semantic productions. For instance, the semantics of the `try` statement has three main productions:

1. a production for `try Block Catch` describing the evaluation of try statements that do not contain a `Finally` clause.
2. a production for `try Block Finally` describing the evaluation of try statements that do not contain a `Catch` clause.
3. a production for `try Block Catch Finally` describing the evaluation of try statements that contain both a `Catch` clause and a `Finally` clause.

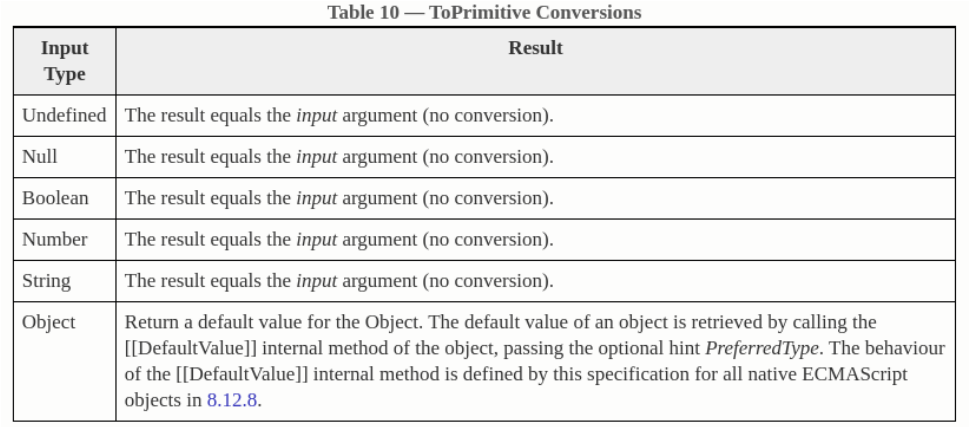


Table 10 — ToPrimitive Conversions

| Input Type | Result |
|---|---|
| Undefined | The result equals the *input* argument (no conversion). |
| Null | The result equals the *input* argument (no conversion). |
| Boolean | The result equals the *input* argument (no conversion). |
| Number | The result equals the *input* argument (no conversion). |
| String | The result equals the *input* argument (no conversion). |
| Object | Return a default value for the Object. The default value of an object is retrieved by calling the [[DefaultValue]] internal method of the object, passing the optional hint *PreferredType*. The behaviour of the [[DefaultValue]] internal method is defined by this specification for all native ECMAScript objects in 8.12.8. |

Figure 6.3: Conversions table present in the `ToPrimitive` abstract operation.

These productions further rely on two auxiliary productions describing the evaluation of the `Catch` clause and the `Finally` clause. Hence, in total, the semantics of the try statement is described by five separate productions. In such cases, instead of a single ordered list element, the standard includes several ordered list elements, each preceded by an HTML paragraph element with the description of the production. For instance, Figure 6.4 shows the structure of the HTML code corresponding to the try statement.

## 6.2 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.
Figure 6.5 illustrates 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 supplied in JSON format;
5. Transform the `ECMA-SL` code into HTML.

**Filter** Our `ECMAScript` interpreter contains functions that are not to be transformed into HTML, of which the most relevant are auxiliary functions that capture implementation-specific behaviors; for instance, functions that mediate programmatic interaction with our internal representation of `ECMAScript` objects. The first step of the code generation process consists of filtering out these functions. More precisely, all functions to be transformed into HTML must contain a code generation directive next to their signature with their corresponding unique identifier in the standard. Functions that do not contain this code generation directive are removed by the filtering process. The syntax and semantics of code generation directives are explained in Subsection 6.3.1.

```
<section id="sec-12.14">
  ...

  <!-- try Block Catch -->
  <p>The production <span class="prod"><span class="nt">TryStatement</span> <span class="geq">:
    </span> <code class="t">try</code> <span class="nt">Block</span> <span class="nt">Catch</
    span></span> is evaluated as follows:</p>

  <ol class="proc"> ... </ol>

  <!-- try Block Finally -->
  <p>The production <span class="prod"><span class="nt">TryStatement</span> <span class="geq">:
    </span> <code class="t">try</code> <span class="nt">Block</span> <span class="nt">Finally</
    span></span> is evaluated as follows:</p>

  <ol class="proc"> ... </ol>

  <!-- try Block Catch Finally -->
  <p>The production <span class="prod"><span class="nt">TryStatement</span> <span class="geq">:
    </span> <code class="t">try</code> <span class="nt">Block</span> <span class="nt">Catch</
    span> <span class="nt">Finally</span></span> is evaluated as follows:</p>

  <ol class="proc"> ... </ol>

  <!-- Catch -->
  <p>The production <span class="prod"><span class="nt">Catch</span> <span class="geq">:</span>
     <code class="t">catch</code> <code class="t">(</code> <span class="nt">Identifier</span> <
    code class="t">)</code> <span class="nt">Block</span></span> is evaluated as follows:</p>

  <ol class="proc"> ... </ol>

  <!-- Finally -->
  <p>The production <span class="prod"><span class="nt">Finally</span> <span class="geq">:</
    span> <code class="t">finally</code> <span class="nt">Block</span></span> is evaluated as
    follows:</p>

  <ol class="proc"> ... </ol>

  ...
</section>
```

Figure 6.4: A fraction of the HTML code corresponding to the `try` statement.

**Normaliser** The algorithms of the ECMAScript standard are implemented in our reference interpreter as ECMA-SL functions. For instance, the internal method `[[GetOwnProperty]]` is implemented as an ECMA-SL function called `GetOwnProperty`. In contrast, the semantic productions that define the behavior of ECMAScript expressions are implemented within the body of the `JS_Interpreter_Expr` ECMA-SL function, which interprets ECMAScript expressions. Likewise, the `JS_Interpreter_Stmt` ECMA-SL function encodes the semantics of ECMAScript statements. Both these functions start with an ECMA-SL match statement that selects the production to be applied. Hence, the productions for ECMAScript expressions and statements are implemented as pattern clauses of an ECMA-SL match statement.

Figure 6.6 shows three semantic productions of the `try` statement and corresponding ECMA-SL code. One can see that each semantic production corresponds to a different pattern clause of the match statement. For instance, the semantic production `try Block Catch` corresponds to the following pattern clause:

```
| { type: "TryStatement", block: Block, handler: Catch, finalizer: null } -> { ... }
```

The ECMA-SL2English tool does not contain code to transform ECMA-SL match statements into HTML. Instead, in order to support match statements, we transform each pattern clause into a single ECMA-SL function. The name of the newly created function is automatically generated and its body is set to the statements of the corresponding pattern clause. By transforming each semantic production into an ECMA-SL function, we make the code generation process uniform in that we use the same code to generate the internal algorithms of the standard and its semantic productions.

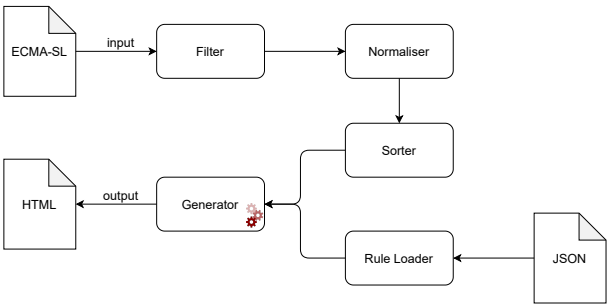



Figure 6.5: `ECMA-SL2English` execution pipeline.

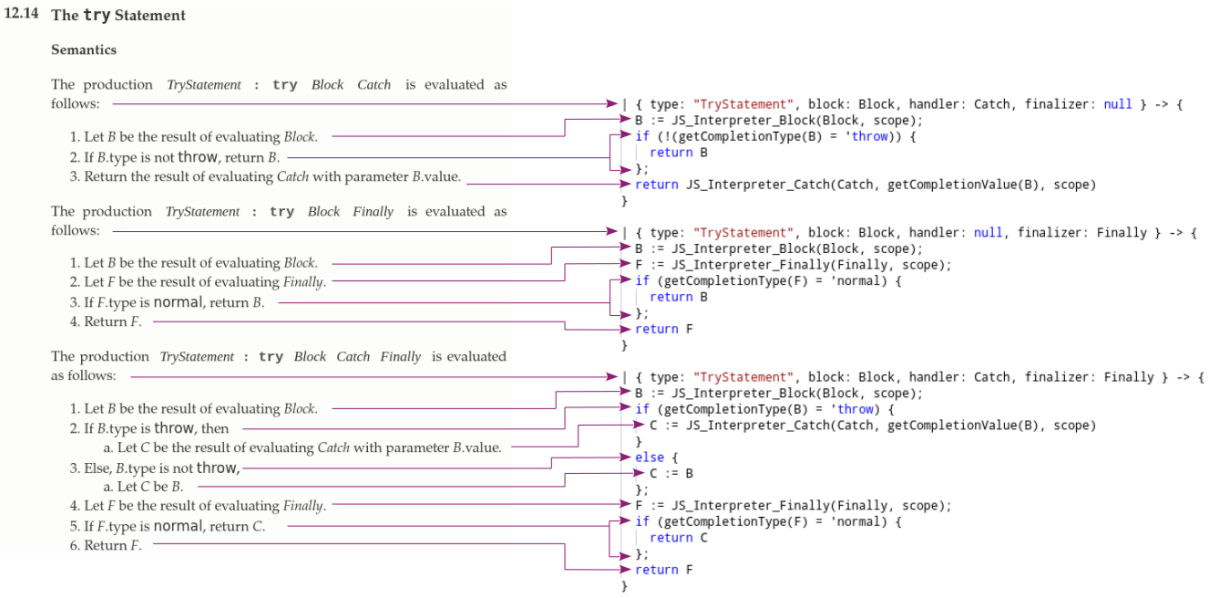



Figure 6.6: `try` statement and corresponding `ECMA-SL` code.

**Sorter** To generate an HTML version of the standard that coincides with the official `ECMAScript` document, we must ensure that the HTML file produced by the `ECMA-SL2English` tool follows the order by which the algorithms appear in the standard. However, the organisation of the `ECMA-SL` functions in our interpreter does not precisely match the organisation of the corresponding algorithms in the standard. Note that the code of our reference interpreter is split into multiple files with each file not necessarily following the exact same order as the standard. So, before the HTML generation step, we need to order the `ECMA-SL` functions according to their respective position in the standard. To this end, we annotate our `ECMA-SL` functions with a code generation directive that specifies their section number; for instance, the `[[Call]]` internal method is annotated with the section number `13.2.1`. Given that all algorithms are annotated with their corresponding section numbers, we simply sort them according to the lexicographic order of their respective numbers; for instance, the `[[Call]]` internal method, which has section number `13.2.1`, appears after the algorithm for *Creating Function Objects*, which has section number `13.2`.

Recall the structure of the official `ECMAScript` HTML document described in 6.1. The organisational elements of the standard follow a hierarchical structure where a sub-sub-section is represented by an HTML element that is a descendant of an HTML element that represents a sub-section which, in turn, is a descendant of an HTML element that represents a section. For instance, the algorithm of the `[[Call]]` internal method is a descendant of the algorithm for *Creating Function Objects*. In `ECMA-SL`, however, functions cannot be placed inside other functions, they are all top-level elements of the same `ECMA-SL` program. Hence, we cannot directly encode the hierarchical structure of the standard into

```
| Assign (x, e) ->
  match ctxt with
  | SameItem ->
    sprintf
      "let <i>%s</i> be %s"
      x
      (E_Expr.to_html Let e)
  | _ ->
    sprintf
      "<li>Let <i>%s</i> be %s.</i>"
      x
      (E_Expr.to_html Let e)
```

(a) OCaml

```
bindings := getBindingObject(envRec);
assert ({bindings.HasProperty}(bindings, N) = false)
    ;
if (D = true) {
  configValue := true
} else {
  configValue := false
};
```

(b) ECMA-SL

2. Let *bindings* be the binding object for *envRec*.
3. Assert: The result of calling the [[HasProperty]] internal method of *bindings*, passing *N* as the property name, is **false**.
4. If D is **true** then let *configValue* be **true** otherwise let *configValue* be **false**.

(c) ES5 standard

Figure 6.7: Code snippet of the HTML generation for variable assignments (a); ECMA-SL code snippet containing three variable assignments (b); Snippet of the ECMAScript standard corresponding to the ECMA-SL code snippet (c).

the syntax of our reference interpreter. In order to reproduce this hierarchical structure, we also rely on section numbers, creating a tree-like intermediate structure that organises ECMA-SL functions per nesting level. This intermediate structure is then fed to the Generator component of our pipeline, which simply replicates it at the HTML level.

**Rule Loader** Loading the HTML rules written in JSON format occurs independently from the previously described steps and before the Generator step. Here, all the rules that apply to function calls, property lookups, or binary operations that were written in JSON format are loaded to the system and stored into three dedicated dictionaries: a dictionary for function call rules; a dictionary for property lookup rules; and another dictionary for binary operations rules.

Code generation rules are used for HTML generation in the Generator step. For instance, when generating the HTML code of a function call expression, we first check if we have a code generation rule for the function being called. To this end, we look for the function identifier in the domain of the dictionary used to store function call rules. If we have a code generation rule for that function, we fetch it from the corresponding dictionary and apply it to the supplied arguments to obtain the corresponding HTML representation.

**Generator** At the HTML generation stage all ECMA-SL functions are already loaded in memory, correctly sorted, and normalised. Furthermore, all the code generation rules are also stored in the corresponding dictionaries. To obtain the HTML code for a given ECMA-SL function, we go through all the statements contained in its body and generate their corresponding HTML code. This is a recursive process since a statement may be composed of other statements, expressions, and/or literal values, which have to be processed and turned into HTML code as part of the HTML generation of the enclosing statement. More concretely, to obtain the HTML code for a given ECMA-SL function, we apply the OCaml function `E_Stmt.to_html` to its body. This function traverses the given ECMA-SL statement recursively, calling itself on the sub-statements of the given statement and calling the OCaml function `E_Expr.to_html` on the sub-expressions. Analogously, the OCaml function `E_Expr.to_html` traverses the given expression recursively calling itself on its sub-expressions.

To illustrate the code generation process, let us consider the ECMA-SL code shown in 6.7-(b). The ap-

```
| If (e, s, None) when is_basic s && swappable s ->
  sprintf
    "<li>%s, if %s.</li>"
    (E_Stmt.to_html ~ctxt:SameItem s)
    (E_Expr.to_html IfGuard e)
```

3. Reject, if *current* is **undefined** and *extensible* is **false**.

(a) Swappable if-then statement with basic then-clause with OCaml code (left) and snippet of the corresponding generated standard (right).

```
| If (e, s, None) when is_basic s ->
  sprintf
    "<li>If %s, %s.</li>"
    (E_Expr.to_html IfGuard e)
    (E_Stmt.to_html ~ctxt:SameItem s)
```

5. If every field in *Desc* is absent, return **true**.

(b) Non-swappable if-then statement with basic then-clause with OCaml code (left) and snippet of the corresponding generated standard (right).

```
| If (e, s, None) ->
  sprintf
    "<li>If %s<ol class=\"block\">%s</ol></li>"
    (E_Expr.to_html IfGuard e)
    (E_Stmt.to_html s)
```

3. If *desc*.[[Configurable]] is **true**
   a. Remove the own property with name *P* from *O*.
   b. Return **true**.

(c) Non-swappable if-then statement with non-basic then-clause with OCaml code (left) and snippet of the corresponding generated standard (right).

```
| If (e, s1, Some s2) when is_let s1 && is_let s2 ->
  sprintf
    "<li>If %s, then %s, else %s.</li>"
    (E_Expr.to_html IfGuard e)
    (E_Stmt.to_html ~ctxt:SameItem s1)
    (E_Stmt.to_html ~ctxt:SameItem s2)
```

4. If *D* is **true**, then let *configValue* be **true**, else let *configValue* be **false**.

(d) Non-swappable if-then statement with non-basic then-clause with OCaml code (left) and snippet of the corresponding generated standard (right).

Figure 6.8: Four different rules for HTML generation of if-then-else statements.

plication of our code generation algorithm to this snippet of code results in the HTML code corresponding to the textual representation given in Figure 6.7-(c). Importantly, this HTML code exactly coincides with the HTML code of the standard. In the following, we will examine the application of the code generation algorithm to this snippet of code.

The statement to be turned into HTML is a block statement composed of a variable assignment, an assert statement, and an if-else statement. The if-else statement contains both a then clause and an else clause, each of them comprising a single variable assignment statement. The variable assignments correspond to lines 1, 4, and 6 of the code snippet in Figure 6.7-(b). The three variable assignments are turned into HTML using the following pattern:

```
Let <variable-name> be <expression HTML>.
```

However, depending on the context where they occur, they may or may not be associated with a separate list item. For instance, the variable assignment in line 1 has its own separate list item, while variable assignments in lines 4 and 6 are included in the list item of their enclosing statement. Note that when a variable assignment has its own list item, the word `let` starts with an upper-case `L`; in contrast, when a variable assignment is included in the list item of its enclosing statement, the word `let` starts with a lower-case `l`.

In Figure 6.7-(a), we give the snippet of the OCaml function `E_Stmt.to_html` that turns variable assignments into HTML. Observe that this function uses a variable `ctxt` to reason about the context in which the statement is to be generated. In our example, we only have to distinguish between the `SameItem` context and the other contexts. The `SameItem` context indicates that the statement is to be generated within the list item of its enclosing statement, while in all other contexts we create a new list item.

While the rules for the HTML generation of variable assignments are relatively straightforward, other `ECMA-SL` statements have substantially more complex HTML generation rules. In particular, the if-then-

else statement has fourteen different rules, four of which are shown in Figure 6.8. Below, we give the intuition behind these rules:

1. **Swappable if-then statement with basic then-clause**: We say that an if-then statement is *swappable* if the HTML code of the then clause is to appear before the HTML code of the if guard. Additionally, we say that a statement is *basic* if it is neither a control flow statement nor a function call. In this case, we generate the HTML of the if-then statement using the pattern:

   ```
   <li>[HTML statement_then], if [HTML expression].</li>
   ```

   Figure 6.8-(a) shows the snippet of the OCaml function that turns swappable if-then statements with a basic then-clause into HTML and an excerpt of the standard generated using this rule.

2. **Non-swappable if-then statement with basic then-clause**: In this case, we generate the HTML of the if-then statement using the pattern:

   ```
   <li>If [HTML expression], [HTML statement].</li>
   ```

   Figure 6.8-(b) shows the snippet of the OCaml function that turns non-swappable if-then statements with a basic then-clause into HTML and an excerpt of the standard generated using this rule.

3. **Non-swappable if-then statement with non-basic then-clause**: In this case, the HTML generated for the then clause of the if-then statement must be enclosed in its own ordered list; accordingly, we generate the HTML of the if-then statement using the pattern:

   ```
   <li>If [HTML expression] <ol class="block">[HTML statement]</ol></li>
   ```

   Figure 6.8-(c) shows the snippet of the OCaml function that turns non-swappable if-then statements with a non-basic then-clause into HTML and an excerpt of the standard generated using this rule.

4. **Non-swappable if-then-else statement with two simple assignments**: This rule applies to if-then-else statements whose then and else clauses consist of a single variable assignment. In this case, the HTML generated for both then and else clauses is included in the list item of the entire if-then-else statement. Figure 6.8-(d) shows the snippet of the OCaml function that turns non-swappable if-then-else statements with two simple assignments into HTML and an excerpt of the standard generated using this rule. Note that the context `SameItem` is passed to the function `E_Stmt.to_html` (lines 5 and 6) guaranteeing that the HTML code generated for the then and else clauses does not introduce new list items. Observe that this rule is used to generate the HTML code of the if-then-else statement given in Figure 6.7-(b) (lines 3-7).

## 6.3 Code Generation 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. In this section, we first describe the code generation directives that we added to the syntax of `ECMA-SL` (Subsection 6.3.1) and then explain the mechanism used to load new code generation rules to our system (Subsection 6.3.2).

```
/* 10.2.1.2.6 ImplicitThisValue() */
/* Object Environment Records return undefined as their ImplicitThisValue
   unless their provideThis flag is true. */
function ImplicitThisValueObject(objectEnvRec) [
  "10.2.1.2.6",
  "Object Environment Records return undefined as their ImplicitThisValue
   unless their provideThis flag is true.",
  "",
  "ImplicitThisValue"
] [
  "objectEnvRec:the object environment record for which the method was invoked"
] {
  /* 1. Let envRec be the object environment record for which the method was invoked. */
  envRec := objectEnvRec;
  ...
};
```

(1) Example of the use of code generation directives in ECMA-SL functions.

```
/* 3. Otherwise, return undefined. */
gen_wrapper ["before:Otherwise, "] {
  return 'undefined
}
```

(2) Example of the use of code generation directives in ECMA-SL wrapper statements.

```
/* a. If HasPrimitiveBase(V) is false, then let get be the [[Get]] internal method of base,
      otherwise let get be the special [[Get]] internal method defined below. */
if (HasPrimitiveBase(V) = false) ["after:then"] {
  get := base["Get"]
} else ["replace-with:otherwise"] {
  get := "Get_internal"
};
```

(3) Example of the use of code generation directives in ECMA-SL if-then-else statements.

```
/* 5. For each element P of names in list order, */
foreach (P : names) ["after: in list order,"] ["P:element"] {
  descObj := {props.Get}(props, P);
  desc := ToPropertyDescriptor(descObj);
  descriptors := l_add(descriptors, [P, desc])
};
```

**(4) For-each directive**

Figure 6.9: Four ECMA-SL code snippets illustrating the use of code generation directives.

### 6.3.1 Code Generation Directives

We extended the syntax of ECMA-SL 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 a high-level 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. As an example, consider the code given in Figure 6.9-(a). Here, we include a snippet of the ECMA-SL function ImplicitThisValueObject, which implements the algorithm ImplicitThisValue of Object Environment Records. This function is annotated with its section number, textual description, and section name. We use the empty string, "", to indicate that the algorithm has no additional notes or details to be appended to its HTML code.

   Additionally, the function signature directive allows us to specify custom textual descriptions for the program variables used in the function. This is illustrated in lines 13 and 14 of the function ImplicitThisValueObject given in Figure 6.9-(a). While line 13 shows the text of the standard, line 14 shows our ECMA-SL implementation. One can see that the formal parameter objectEnvRec should be turned into the text "the object environment record ...". To achieve this, we associate the parameter objectEnvRec with the appropriate string in line 11, instructing the HTML code generator to replace its default HTML text with the specified one.

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. As an example, consider the code given in Figure 6.9-(b). The text of the standard should be `Otherwise, return undefined`. However, the `ECMA-SL` statement in our reference implementation is simply `return 'undefined`. If no additional information is provided, it is not possible to infer that the generated text should include the word `Otherwise`. To account for this, we enclose our return statement within a statement wrapper directive that explicitly instructs the code generator to add the required word.

   Some `ECMA-SL` statements of our reference implementation do not have direct counter-parts in the official text of the standard. For instance, in some cases our implementation uses additional statements to update our internal representations of the standard types. In such cases, we make use of a statement wrapper directive that instructs the code generator to ignore the enclosing statement. More concretely, we use the syntax `gen_wrapper [print_ignore] { s }` to ignore the statement `s` during HTML code generation.

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 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. As an example, consider the code given in Figure 6.9-(c). The then directive in line 3 instructs the HTML generator to append the word ”then” to the HTML code generated for the if guard, which normally does not include it. The else directive in line 5 instructs the HTML generator to replace the word ”else” with the word ”otherwise”, guaranteeing that the generated version of the `ECMAScript` standard is faithful to the official one.

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 keyword values are `before` and `after`. In addition, this directive allows us to prepend extra HTML code to that of the iterating variable. As an example, consider the code given in Figure 6.9-(4). The default HTML code generated for this for-each statement is: `For each P of names`. Using the for-each directive, we improve the generated HTML in two ways. First, we add the string ”in list order,” to the end of the for-each guard. Second, we refer to variable `P` as ”element P” instead of simply ”P”. With these two directives, we get the text ”For each element P of names in list order”, which exactly coincides with the text of the standard.

5. **Match pattern directive** The match pattern directive is used to annotate pattern clauses of the `ECMA-SL` match statements. 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.

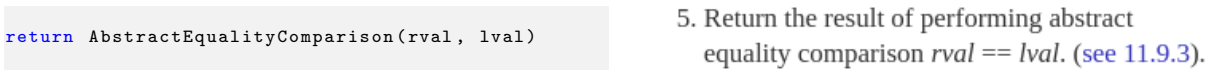



Figure 6.10: ECMA-SL code snippet (left) and corresponding ECMAScript standard HTML text (right).

### 6.3.2 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. For instance, Figure 6.10 shows, on the left, an `ECMA-SL` return statement with a call to the `AbstractEqualityComparison` function. On the right, it shows the corresponding standard HTML text.

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.

We specify a rule template by giving its list of static phrases and the order in which the dynamically generated elements are to be inserted between those phrases. For instance, the template above consists of the static phrases: (1) *"the result of performing abstract equality comparison "* and (2) *" == "*, and the insertion order should be 0 and 1, meaning that the text of the first argument is to be inserted between the two static phrases and the text of the second argument is to be placed at the end. Note that, for some functions, the order of the arguments changes; for instance, the text of the first argument may not correspond to the first placeholder.

In the following, we explain how we represent each type of rule in JSON format by appealing to three illustrative examples.

**Function call rule** Figure 6.11 presents the rule for the function `isUninitialisedBinding` and shows how it is used to generate a concrete fragment of the standard. The function `isUninitialisedBinding` is an auxiliary function of `ECMARef5` that interacts with our internal representation of Environment Records. Hence, it has no direct counter-part in the `ECMAScript` standard, given that the standard is agnostic with respect to the particular data structures used to implement Environment Records. More concretely, this function checks whether or not the Environment Record `envRec` contains an immutable binding for variable `N`. While the standard uses the phrase *"If the binding for N in envRev is an uninitialised immutable binding"*, `ECMARef5` simply calls the function `isUninitialisedBinding`.

To make sure that the generated text of the standard coincides with the original one, we make use of a function call rule that instructs the code generator to produce the matching text as given by the template of Figure 6.11-(a). Note that the arguments of the function call appear in reverse order in

```
the binding for {1} in {0} is an
uninitialised <a
href=\"#immutable-binding\">
immutable binding</a>
```

(a) Template string corresponding the code generation rule in (b).

```
{
  "func_name": "isUninitialisedBinding",
  "phrases": [
    "the binding for ",
    " in ",
    " is an uninitialised <a
    href=\"#immutable-binding\">immutable
    binding</a>"
  ],
  "active_params": [ 1, 0 ]
}
```

(b) Code generation rule in JSON format corresponding to function calls to `isUninitialisedBinding`

3. If the binding for *N* in *envRec* is an uninitialised immutable binding, then
    a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
4. Else, return the value currently bound to *N* in *envRec*.

(c) ECMAScript standard HTML text corresponding to the ECMA-SL code snippet in (d).

```
if (isUninitialisedBinding(envRec, N)) {
  if (S = false) {
    return 'undefined
  };
  throw ReferenceErrorConstructorInternal()
}
else {
  return getBindingValue(envRec, N)
}
```

(d) ECMA-SL code snippet showing a call to the function `isUninitialisedBinding`.

Figure 6.11: Example of code generation rule applied to a function call, including string template (a), rule in JSON (b), HTML of the standard (c), and the ECMA-SL code (d).

```
{0} is greater than {1}
```

(a) Template string corresponding the code generation rule in (b).

```
{
  "oper": ">",
  "phrases": [
    "",
    " is greater than ",
    ""
  ],
  "expressions_order": [ 0, 1 ]
}
```

(b) Code generation rule in JSON format corresponding to the binary operator "greater than".

ii. If *n* is greater than *argCount*, let *v* be **undefined**.

(c) ECMAScript standard HTML text corresponding to the ECMA-SL code snippet in (d).

```
if (n > argCount) {
  v := 'undefined
}
```

(d) ECMA-SL code snippet showing a "greater than" binary operation.

Figure 6.12: Example of code generation rule applied to the binary operation "greater than", including string template (a), rule in JSON (b), HTML of the standard (c), and the ECMA-SL code (d).

the rule template; the second argument is the first to appear and the first argument the last. This rule template is represented by the JSON rule given in Figure 6.11-(b). This rule is composed of:

- The name of the function to which it applies, `isUninitialisedBinding`.
- Three static phrases: (i) *"the binding for "*; (ii) *" in "*; and (iii) *" is an uninitialised ..."*.
- A list specifying which parameter is to be inserted between each two consecutive static phrases; hence, the number of active parameters must be equal to the number of static phrases minus one. As we have three static phrases, we must have two active parameters describing which parameters are to be inserted between the first and second phrases and the second and third phrases. Note that the same parameter may appear multiple times.

Finally, Figures 6.11-(c) and 6.11-(d) show the generated snippet of the standard and its corresponding `ECMA-SL` implementation.

```
the value of the [[{1}]]
internal property of {0}
```

(a) Template string corresponding the code generation rule in (b).

```
{
  "prop_name": [
    "Extensible",
    "Prototype"
  ],
  "active_elements": [ 1, 0 ],
  "phrases": [
    "the value of the [[",
    "]] internal property of "
  ]
}
```

(b) Code generation rule in JSON format corresponding to property lookups on the properties `Extensible` and `Prototype`.

4. If *proto* is **null**, then return the value of the [[Extensible]] internal property of *O*.

(c) ECMAScript standard HTML text corresponding to the ECMA-SL code snippet in (d).

```
if (proto = 'null) {
  return O.Extensible
}
```

(d) ECMA-SL code snippet showing a property lookup of `Extensible` in object `O`.

Figure 6.13: Example of code generation rule applied to the binary operation "greater than", including string template (a), rule in JSON (b), HTML of the standard (c), and the ECMA-SL code (d).

**Operator rule** Figure 6.12 presents the rule for the binary operator "greater than" and shows how it is used to generate a concrete fragment of the standard. Most of the operations in the `ECMAScript` standard have a textual representation that is not equal to their corresponding mathematical symbol. For instance, the standard "greater than" operator, typically represented by the symbol ">", appears in the standard in words; instead of "`x > y`", the standard uses "`x` is greater than `y`".

To make sure that the generated text of the standard coincides with the original one, we make use of an operator rule that instructs the code generator to produce the matching text as given by the template in Figure 6.12-(a). This rule template is represented by the JSON rule given in Figure 6.12-(b), which is composed of:

- The `ECMA-SL` operator to which it applies, `>`.
- Three static phrases: (i) ""; (ii) *" is greater than "*; (iii) "".
- A list specifying the order in which the left- and right-hand side expressions are to be inserted in between the static phrases.

Analogously to function call rules, the number of static phrases of an operator rule is equal to the arity of the operator plus one. The "active_parameters" specify the order in which the HTML code of the parameters is to be inserted in between the static phrases. Finally, Figures 6.12-(c) and 6.12-(d) show the generated snippet of the standard and its corresponding `ECMA-SL` implementation.

**Property lookup rule** Figure 6.13 presents the rule for property lookups and shows how it is used to generate a concrete fragment of the standard. The default textual representation for a property lookup uses the dot-notation; for instance, accessing the attribute `[[Configurable]]` of a property descriptor `Desc` is presented in the standard as `Desc.[[Configurable]]`. However, there are various exceptions. Most notably, accesses to internal properties of `ECMAScript` objects are usually represented according to the template given in Figure 6.13-(a). For example, instead of writing `O.[[Extensible]]`, the standard uses the phrase *"the value of the [[Extensible]] internal property of O."*

To make sure that the generated text of the standard coincides with the original one, we make use of a property lookup rule that instructs the code generator to produce the matching text as given by the template in Figure 6.13-(a). This rule template is represented by a JSON rule given in Figure 6.13-(b). This rule is composed of:

- The name of the property to which it applies. Note that a list of properties is also acceptable, making this rule applicable to all the specified properties.
- Three static phrases: (i) ”the value of the [[”; (ii) ”]] internal property of ”; (iii) ””.
- A list specifying the order in which the object and the property expressions are inserted when concatenating the static phrases.

Note that an empty string is provided in the list of static phrases as the last element. This means that no extra text is added after the generated HTML for the property expression.

# Chapter 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`. Hence, we structure this chapter into two sections with the first one covering `ECMARef5` and the second one `ECMA-SL2English`.

## 7.1 ECMARef5 Evaluation

`ECMARef5` was tested against `Test262` [5], the official `ECMAScript` test suite. This test suite provides a set of non-normative software tests used to help validate conforming `ECMAScript` language implementations. `Test262` is routinely used by developers of `ECMAScript` engines to test their `ECMAScript` implementations. As the `ECMAScript` language is in constant evolution, `Test262` also has to evolve to cover the new features of the language. `Test262` is comprised of thousands of test files, often including multiple test cases per test file. The size of test files varies substantially, ranging from a few lines to hundreds of lines[1].

```
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: 11.13.1-4-28gs
6  description: >
7      Strict Mode - TypeError is thrown if the identifier 'Math.PI'
8      appears as the LeftHandSideExpression of simple assignment(=)
9  negative: TypeError
10 flags: [onlyStrict]
11 ---*/
12
13 Math.PI = 20;
```

(a) 11.13.1-4-28gs.js

```
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: 15.2.3.13-2-1
6  description: Object.isExtensible returns true for all built-in objects (Global)
7  ---*/
8
9  var global = this;
10
11 assert(Object.isExtensible(global));
```

(b) 15.2.3.13-2-1.js

Figure 7.1: Contents of two `Test262` test files.

Figure 7.1 shows two `Test262` files. Each test is composed of three distinct sections: (1) *copyright section* with information regarding the authors of the test; (2) *frontmatter section* with the metadata of the test; and (3) *body section* with the code of the test. The frontmatter comprises various components describing different aspects of the test file, such as:

- `description`: a short one-line description of the purpose of the test;
- `esid`: the version of the standard targeted by the test;
- `negative`: a keyword indicating whether or not the test is supposed to throw an error; and

[1] e.g. 328 lines in test file `language/expressions/left-shift/S11.7.1_A5.2_T1.js`

- `flags`: a list of boolean properties associated with the test, of which the most relevant to us is `onlyStrict` that is used to indicate whether or not the test is to be run in strict mode.

Considering the two examples given in Figure 7.1, the left-hand side test targets Section 11.13.1 of the 5th edition of the standard, must be run in strict mode, and is supposed to throw an error of type `TypeError`. In contrast, the right-side test targets Section 15.2.3.13, must not be run in strict mode, and is not supposed to throw an error during execution.

`Test262` comes with a number of auxiliary functions to be used in test cases. For instance, the function `assert(e)` is used to check whether or not `e` evaluates to true; the function `isEqual(num1, num2)` tests if two numbers denote the same value; and the function `compareArray(a, b)` checks whether two arrays have the same length and, if so, if they have equal values at equal indexes. These functions are all organised in a collection of files referred to as the *harness* of `Test262`. Hence, in order to run any `Test262` test, one needs to include the code of the harness. In our project, we simply prepend the code of the harness to the code of the test to be executed.

### 7.1.1 Test selection

In its current version, `Test262` targets the most recent edition of the `ECMAScript` standard, ES11, and comprises ≈40k test files. As mentioned before, test files are annotated with the version of the standard that they target. The `ECMAScript` committee has striven to maintain retro-compatibility between older versions and more recent versions of the standard. Hence, tests targeting `ECMAScript` 5 should exhibit the same behaviour in `ECMAScript` engines that support at least the fifth version of the standard. There are, however, exceptions. In the sixth version of the standard, the length property of function objects is configurable, while in the fifth it was not. For all exceptions, we have chosen to follow the behaviour described in the more recent versions of the standard, meaning that, if there is, for instance, an incompatibility between ES5 and ES6, we will implement the behaviour described in ES6.

| **ES5 Tests** | **12,186** |
|---|---|
| Misclassified | 58 |
| Unimplemented features | 54 |
| Timed-out | 6 |
| **Applicable Tests** | **12,068** |
| Passed Tests | 12,026 |
| Failed Tests | 42 |

Table 7.1: Breakdown of the Test262 tests.

The breakdown of the test filtering process is presented in Table 7.1. The version of `Test262` used in our evaluation contains 12,186 test files that target the fifth version of the standard. From these, we removed 118 tests: 58 tests that are misclassified, are annotated as targeting the fifth version of the standard but that we identified as testing features of the sixth version of the standard; 54 tests that target features of the language that are not yet implemented in `ECMARef5`, for instance, function `decodeURIComponent`; and, 6 tests that timed-out during tests execution. This leaves us with 12,068 applicable tests of which we pass 99.6%, giving us a strong guarantee that `ECMARef5` implements the ES5 standard correctly.

### 7.1.2 Testing pipeline

We are now at the position to explain our testing pipeline. We will first present a non-optimised pipeline, which is easier to understand, and only then present the optimised pipeline that we have used to test our tool. Importantly, with the included optimisations, we gained a 295% performance boost.

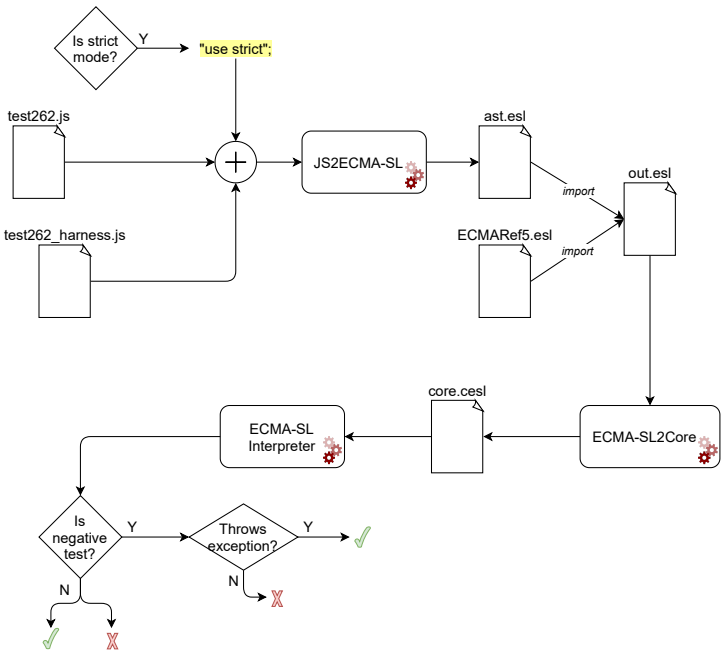



Figure 7.2: Test262 test execution pipeline.

#### Non-optimised testing pipeline

Figure 7.2 illustrates the non-optimised testing pipeline. Before compiling the `ECMAScript` program to `ECMA-SL`, both the source code of the test file and the source code of the testing harness must be concatenated. If the test is to be run in strict mode, the test code must also be prepended with the directive `"use strict"`, which activates strict mode. A test is to be run in strict mode if the property `onlyStrict` is included as a flag in the frontmatter section of the test. The resulting `ECMAScript` program is then given to the `JS2ECMA-SL` compiler, which generates an `ECMA-SL` file, `ast.esl`, with the compilation of the test. Recall that this file simply contains an `ECMA-SL` function returning the `ECMA-SL` object graph representing the input program. This file is then combined with the `ECMARef5` written in `ECMA-SL`, `ECMARef5.esl`, to obtain the final compiled program, `out.esl`.

In the next phase, the obtained `ECMA-SL` file is compiled to `Core ECMA-SL`, generating the `core.cesl` file. Note that all the imports included in the file `out.esl` are resolved as part of the compilation to `Core ECMA-SL`. Hence, the returned program, `core.cesl`, is completely self-contained, including all the code that is required for the execution of the test in a single file.

Finally, we interpret the obtained `Core ECMA-SL` program using our `ECMA-SL` interpreter. The output of the interpreter is then compared against the expected output of the test. `Test262` includes both positive and negative tests; positive tests are expected to execute normally, while negative tests are expected to throw an exception. A test is negative if the property negative is included as a flag in the frontmatter section of the test file. Hence, in order to validate the test result, we first check whether or not the test is negative. If the test is negative, we check if the interpretation of the code throws an exception and if the thrown exception has the same type as the one identified by the key `negative`. If the test is positive, we

simply check if the interpretation completes successfully.

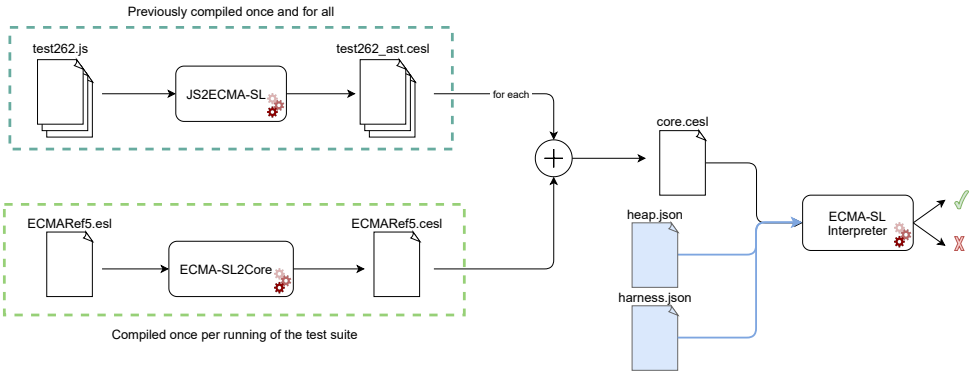



Figure 7.3: Test262 test fully optimised execution pipeline.

### Optimised testing pipeline

Running the non-optimised pipeline on all selected tests takes approximately 3 hours and 28 minutes. This delay renders the development process cumbersome, as we have to run all tests regularly to make sure that we have not introduced new bugs in `ECMARef5`. This is especially important when modifying core functionalities, such as internal functions and auxiliary code in general. In order to streamline the testing process, we have developed an optimised version of the testing pipeline introduced above. Figure 7.3 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. Recall that each compiled test file simply contains a `Core ECMA-SL` function that generates the object graph corresponding to the AST of the respective test. 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. Recall that one of the first steps of the interpretation of an `ECMAScript` program is to create the initial heap on which that program is to be evaluated. This initial heap contains all the built-in objects and their corresponding function objects. Unsurprisingly, the creation of the initial heap involves the execution of thousands of `ECMA-SL` commands ($\approx$ 454k) , taking a significant amount of time when compared to the time taken by the actual execution of the test. 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.

| Section | #T | Passed | Failed | Opt. Time | Non-Opt. Time |
|---|---|---|---|---|---|
| 7 (Lexical Conventions) | 545 | 543 | 2 | 02m36s | 08m10s |
| 8 (Types and Objects Internal Methods) | 184 | 184 | 0 | 00m55s | 03m00s |
| 9 (Type Conversion and Testing) | 115 | 115 | 0 | 00m30s | 01m50s |
| 10 (Execution Code and Contexts) | 414 | 413 | 1 | 02m08s | 06m49s |
| 11 (Expressions) | 1635 | 1634 | 1 | 09m20s | 27m56s |
| 12 (Statements) | 648 | 648 | 0 | 03m16s | 10m12s |
| 13 (Function Definition) | 228 | 228 | 0 | 01m12s | 03m42s |
| 14 (Program) | 24 | 24 | 0 | 00m06s | 00m23s |
| 15.1 (Global) | 195 | 195 | 0 | 01m48s | 04m05s |
| 15.2 (Object) | 2885 | 2885 | 0 | 14m35s | 47m44s |
| 15.3 (Function) | 411 | 410 | 1 | 02m22s | 07m02s |
| 15.4 (Array) | 2268 | 2268 | 0 | 15m12s | 41m21s |
| 15.5 (String) | 861 | 856 | 5 | 04m24s | 14m14s |
| 15.6 (Boolean) | 34 | 34 | 0 | 00m09s | 00m32s |
| 15.7 (Number) | 191 | 174 | 17 | 00m53s | 03m04s |
| 15.8 (Math) | 171 | 171 | 0 | 01m14s | 03m10s |
| 15.9 (Date) | 539 | 533 | 6 | 02m44s | 08m55s |
| 15.10 (RegExp) | 520 | 513 | 7 | 05m56s | 11m52s |
| 15.11 (Error) | 84 | 84 | 0 | 00m26s | 01m23s |
| 15.12 (JSON) | 116 | 114 | 2 | 00m40s | 02m00s |
| Total (this thesis) | 7764 | 7742 | 22 | 41m30s | 2h09m02s |
| Total | 12068 | 12026 | 42 | 1h10m26s | 3h27m44s |

Table 7.2: Test262 testing results per section of the ECMAScript standard.

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. Hence, we can load its AST directly to memory from a pre-generated JSON serialisation.

The speedups obtained using the proposed pipeline are given in the following subsection together with the breakdown of the testing results.

### 7.1.3 Testing results

Table 7.2 presents the breakdown of the testing results per section of the ECMAScript standard, with the exception of Section 15 for which it presents the results at the subsection granularity, as this section contains the specification of all the ECMAScript built-in objects and we chose the give our results for each built-in object separately. 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.

| Section | #T | Executed Commands | | |
|---|---|---|---|---|
| | | Min | Max | Mean |
| 7 (Lexical Conventions) | 545 | 1413 | 1418962 | 103965 |
| 8 (Types and Objects Internal Methods) | 184 | 718 | 19947460 | 155821 |
| 9 (Type Conversion and Testing) | 115 | 1632 | 243506 | 24097 |
| 10 (Execution Code and Contexts) | 414 | 1413 | 868626 | 54300 |
| 11 (Expressions) | 1635 | 1345 | 957352 | 37924 |
| 12 (Statements) | 648 | 1413 | 552813 | 38220 |
| 13 (Function Definition) | 228 | 1413 | 673708 | 36791 |
| 14 (Program) | 24 | 7665 | 54763 | 21992 |
| 15.1 (Global) | 195 | 2362 | 150983 | 43841 |
| 15.2 (Object) | 2885 | 2635 | 2775198 | 152295 |
| 15.3 (Function) | 411 | 2617 | 216521 | 41835 |
| 15.4 (Array) | 2268 | 2362 | 147320457 | 578229 |
| 15.5 (String) | 861 | 2362 | 779023 | 32418 |
| 15.6 (Boolean) | 34 | 2635 | 150641 | 25437 |
| 15.7 (Number) | 191 | 2635 | 275352 | 43131 |
| 15.8 (Math) | 171 | 3352 | 22656750 | 611864 |
| 15.9 (Date) | 539 | 2635 | 665132 | 49216 |
| 15.10 (RegExp) | 520 | 2362 | 311306551 | 675194 |
| 15.11 (Error) | 84 | 2661 | 2179770 | 129620 |
| 15.12 (JSON) | 116 | 12775 | 1486922 | 270168 |
| Total | 12068 | - | - | - |

Table 7.3: Test262 testing results per section of the ECMAScript standard with information about the number of Core ECMA-SL executed commands.

Table 7.2 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. For instance, the non-optimised execution time for the Array built-in object is 41 minutes and 21 seconds, while the optimised execution time is 15 minutes and 12 seconds. 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.

Finally, the purpose of Table 7.3 is to show the number of Core ECMA-SL commands executed per section. Hence, for each section of the standard, we give its total number of tests together with the minimum, maximum, and average numbers of executed Core ECMA-SL commands. For instance, Section 15.2 contains 2,885 test files and, on average, each test file executes 152,295 Core ECMA-SL commands. Furthermore, the shortest-running test executes 2,635 commands and the longest-running one executes 2,775,198 commands. It is clear that even small ECMAScript programs trigger the execution of thousands of Core ECMA-SL commands. This is due to the fact that the ECMARef5 interpreter performs all the steps established by the ECMAScript standard without applying any simplifications/optimisations.

## 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 [43] as well as HTML-specific metrics based on the concept of tree similarity [7]. For both classes of metrics, we rely on existing out-of-the-box open-source implementations; for text-based comparison metrics we have used the `textdistance` open-source Python project [6], and for HTML-specific metrics we have used the `HTMLSimilarity` open-source Python project [44]. Below, we briefly describe these two classes of comparison metrics.

**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 [45]. The edit distance between two sequences of characters measures the similarity of the two given sequences as the number of edit operations needed to turn one sequence into another. Typically, the edit distance operations are: (1) character insertion; (2) character deletion; and (3) character replacement. For instance, the edit distance between `abb` and `abc` is 1 (replace `b` for `c` in the first string) and the edit distance between `abb` and `ab` is also 1 (remove `b` in the first string). In our context, the edit distance between the generated standard and the official one gives us the number of edit operations that are required to turn the generated standard into the official one or vice-versa.

**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* [46] and the idea is to find the longest contiguous matching subsequence. The style similarity calculates the *jaccard similarity coefficient* [47] between the CSS classes existing in both HTML documents.

**Scope and Granularity** Before proceeding to the presentation of the results, we should clarify the scope and granularity of the evaluation. The HTML generator was only applied to the parts of `ECMARef5` developed in the context of this thesis (recall that the `ECMA-SL` implementations of the built-in objects Array, String, Date, JSON, and RegExp were developed as part of other projects). 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 work. Here we focus on the core sections and built-in objects of the standard.

Analogously to the `Test262` results, the code generator results are also presented at the granularity of the sections of the standard, with the exception of Section 15, whose results are given at the subsection granularity. Hence, in the following, we present the results of the comparison algorithms for Sections 8, 9, 10, 11, 12, 13, 14, 15.1, 15.2, and 15.3.

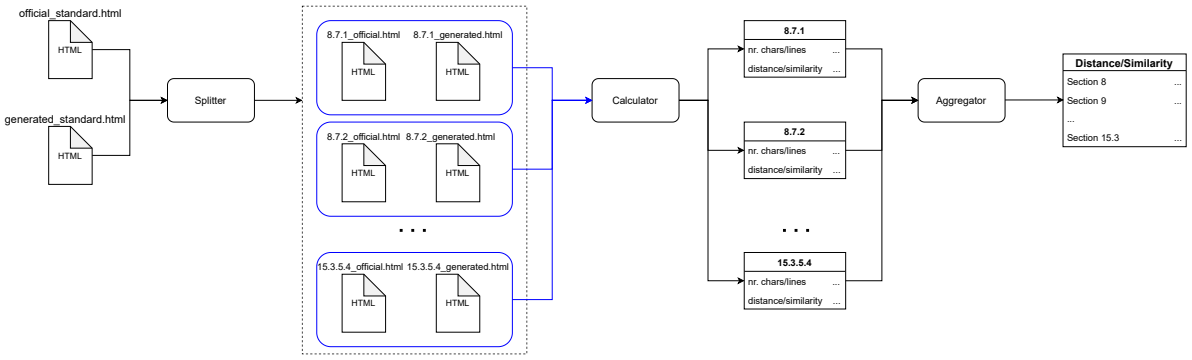



Figure 7.4: Pipeline used to calculate the HTML similarity and text distance between generated and official version of the `ECMAScript` standard.

### 7.2.1 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. For instance, Section 9 is comprised of Subsections 9.1 to 9.12, each consisting of a single algorithm of the standard. Hence, we start by splitting the generated and the official versions of Section 9 into the corresponding subsections; then, we apply the text-comparison algorithms to each pair of subsections (`generated_9.1` vs `official_9.1`, ..., `generated_9.12` vs `official_9.12`) obtaining the corresponding similarity measures; finally, we compute the weighted average of the similarity measures, assigning to each pair the weight corresponding to the size of the official subsection. The complete evaluation pipeline is shown in Figure 7.4 and 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. For each section $Sec_i$ and similarity measure $M_k$, we compute the similarity between the official and generated sections according to the measure $M_k$ using the formula below:

$$S_i^k = \frac{\sum_{j=1}^{n} SS_{ij}^k . L_{ij}}{\sum_{j=1}^{n} L_{ij}} \tag{7.1}$$

   Where $SS_{ij}^k$ denotes the computed similarity for the $j$-th subsection of section $i$ using the measure $M_k$ and $L_{ij}$ denotes the number of lines of the $j$-th subsection of section $i$ in the official version of the standard.

### 7.2.2 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` Python open-source project [6]. This project comes with nine different variations of the edit distance algorithm, of which we use the following six:

- **Levenshtein**: the Levenshtein distance metric is the most classical edit distance metric. It defines the distance between the two given strings by counting the number of edit operations needed to transform one into the other. The allowed edit operations are: insertion, deletion, and substitution.

- **Jaro-Winkler**: the Jaro-Winkler distance metric is an extension of the Jaro distance metric, which gives more favorable scores to strings that match from the beginning. More precisely, the Jaro-Winkler distance requires the common characters of both strings to appear in the same order and within a certain distance of each other. To illustrate, consider the examples below:

```
>> textdistance.jaro_winkler("apple", "appl")
0.96
>> textdistance.jaro_winkler("apple", "app")
0.87
>> textdistance.jaro_winkler("apple", "ple")
0.51
```

```
>> textdistance.levenshtein("apple", "appl")
0.8
>> textdistance.levenshtein("apple", "app")
0.6
>> textdistance.levenshtein("apple", "ple")
0.6
```

  In the first two cases, the Jaro-Winkler algorithm returns a high score as the strings match from the beginning having, respectively, one and two characters missing. In the third case, the two strings do not match from the beginning, resulting in a lower Jaro-Winkler similarity score. As expected, the scores obtained for the Levenshtein distance metric are lower than those obtained for the Jaro-Winkler distance metric in the first two cases and higher in the last one.

- **Needleman-Wunsch**: the Needleman-Wunsch distance metric is a distance metric originally designed to match large DNA sequences. Unlike the previous metrics, this metric is customisable, allowing for the specification of a penalty factor for matching gaps. This penalty factor is relevant for bioinformatic applications but here we set it to 1. Hence, the results we obtained for the Needleman-Wunsch distance metric almost coincide with those obtained for the Levenshtein distance metric.

- **Smith-Waterman**: the Smith-Waterman distance metric is a variation of the Needleman-Wunsch distance metric that allows for the definition of a variable penalty factor for matching gaps. While the Needleman-Wunsch metric uses a constant penalty factor, the Smith-Waterman metric allows for the specification of penalty factors that vary with the size of the matching gap. This means that one can configure the Smith-Waterman metric to prioritise local matches over sparse global matches.

- **Gotoh**: the Gotoh distance metric is yet another variation of the Needleman-Wunsch distance metric which uses a different cost model for matching gaps. In particular, the chosen cost model stems from the assumption that a single large matching gap is biologically more likely to occur than a large number of smaller matching gaps interspersed with matching elements.

- **strcmp95**: the strcmp95 distance metric is an alternative implementation of the Jaro-Winkler distance metric; hence, the obtained results for these two metrics almost exactly coincide.

Below we explain the reasons for excluding the following edit distance metrics:

| Section | Levenshtein | Jaro-Winkler | Strcmp95 | Needleman-Wunsch | Gotoh | Smith-Waterman |
|---|---|---|---|---|---|---|
| 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 7.4: Results of the application of some Edit Distance algorithms to sections of the ECMAScript standard generated by the ECMA-SL2English tool.

- **Hamming**: the Hamming distance metric compares two strings character-by-character, ignoring possible matching (mis-)alignments. This metric is entirely inadequate for our setting given that the generated and the official versions of the standard often use words with different lengths. This results in mis-alignments between the two texts that would cause the Hamming distance metric to output very low values. Note that it suffices for one word in the generated text to have one extra character than the corresponding word in the official text for their similarity to be highly affected.
- **Damerau-Levenshtein**: we have run the Damerau-Levenshtein algorithm on our datasets, obtaining exactly the same results as those obtained by the Levenshtein algorithm. Hence, we chose not to present these results.
- **MLIPNS**: the MLIPNS distance metric was designed to compute the similarity between short character strings corresponding to commercial product names. This metric is not applicable to our setting because it was not designed to deal with large fragments of text consisting of thousands of words, such as the text of the ECMAScript standard.

**Results** We present the overall results for the six selected metrics in Table 7.4. 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%). Below, we clarify the reasons for the lower results obtained for these sections:

1. In Section 14, ECMARef5 slightly deviates from the pseudo-code of the official standard. For instance, it does not include the initial step required to determine whether or not the code is to be executed in strict mode. Given that this is a particularly small section, with only 298 lines in total, even small deviations have a considerable impact on the overall similarity score.
2. In Section 9, the low overall score is caused by the low score obtained for Subsection 9.10, which is below 50%. This subsection describes the abstract operation CheckObjectCoercible which is used to check if the given argument can be converted into an object. This abstract operation is described in tabular form. However, in contrast to all other HTML tables of Section 9, for which

| Section | Style | Structural | Joint | #Lines Official | #Lines 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 7.5: Results of the application of HTML similarity to sections of the ECMAScript standard generated by the ECMA-SL2English tool.

only the header cells have an in-lined custom CSS style, all cells of this table have a similar in-lined custom CSS style. This is a particularly poor design choice since the applied CSS style has no visible effect on the associated table cells. We could have easily ”fixed” this difference by generating this specific table with the required CSS style applied to all cells. If we were to do that, the obtained similarity score for the Subsection 9.10 would increase from ≈50% to ≈93% and the overall score would increase from ≈74% to ≈85%.

### 7.2.3 HTML-specific Metrics

Besides the text-based metrics discussed in the previous subsection, we have used HTML-specific metrics based on the concept of tree similarity [7]. More specifically, we have used the HTMLSimilarity open-source Python project [44] to compute the structural similarity and the style similarity between the official and the generated versions of the standard. Results are presented in Table 7.5. 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. More concretely:

1. Section 10 is affected by the structural similarity scores calculated for Subsections 10.5 and 10.6. Both these subsections represent approximately half of the total number of lines of Section 10.
2. Section 12 is affected by the structural similarity score calculated for Subsection 12.11, which represents one forth of the total number of lines of Section 12.

The common characteristic of these three 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 subsections of the standard.

# Chapter 8

# Conclusions

As the ECMAScript standard becomes more complex, it also becomes more difficult to manage and extend. We believe that programming language tools should play a central role in the ECMAScript standardisation process in order to guarantee that newly added features maintain the internal invariants of the ECMAScript standard and, most importantly, do not break the behavior of existing features. 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 ECMAScript 5 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 ECMAScript 5 standard using out-of-the-box text-based comparison metrics, thereby improving on the *eyeball closeness* methodology of the JSCert project [12].

The two main outcomes of this thesis, ECMARef5 and ECMA-SL2English, were thoroughly evaluated. ECMARef5 was tested against *Test262* [5] passing 12,026 tests out of 12,068 applicable tests. Importantly, ECMARef5 is the most complete reference implementation of the ECMAScript 5 standard, with JS-2-JSIL [8], the second most complete, only passing 8797 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).

The software deliverables of this thesis will be open-sourced and made available online in the future, together with the remaining modules of the ECMA-SL project.

**Future work** We highlight two types of future work: immediate and long-term. Due to the time-frame of the project, we were not able to apply the ECMA-SL2English HTML generator to the entire ECMARef5 implementation. Therefore, our immediate future work would be to extend ECMARef5 with the code generation directives and rules necessary to obtain the HTML code of the missing sections of the standard (15.4 to 15.12). In the long term, we would like to adapt our ECMARef5 implementation to the more recent versions of the ECMAScript standard and to leverage it to semi-automatically obtain various other types

of tools, such as:

- A compiler from ECMAScript to an existing intermediate language for ECMAScript analyses, such as JSIL [8]. The idea would be to first compile the input program to Core ECMA-SL by partially evaluating ECMARef5 on the input program and then compile the resulting Core ECMA-SL program to JSIL using a custom-made compiler.
- A test suite synthesizer that would generate a conformance test suite for the ECMAScript standard from the code of ECMARef5. The idea would be to apply dynamic analyses techniques [11] to the code of ECMARef5 to obtain a set of input programs that would maximise the coverage of the standard.
- A tool that would generate a reference interpreter written in ECMA-SL from the text of the ECMAScript standard.

# Bibliography


[1] "Ecmascript® language specification, 5.1 edition / june 2011." http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf. Accessed on 2020-06-07.

[2] 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.

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

[4] S. McKenzie, "Babeljs - a free and open-source javascript transcompiler." https://babeljs.io. Accessed on 2021-07-30.

[5] "Test262 - official ecmascript conformance test suite." https://github.com/tc39/test262. Accessed on 2020-06-07.

[6] "Textdistance - python library for comparing distance between two or more sequences by many algorithms." https://github.com/life4/textdistance. Accessed on 2021-07-13.

[7] H. Xu, "An algorithm for comparing similarity between two trees," Master's thesis, Duke University, April 2014.

[8] 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.

[9] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," pp. 126–150, 06 2010.

[10] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science, Prentice Hall, 1993.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005* (V. Sarkar and M. W. Hall, eds.), pp. 213–223, ACM, 2005.

[12] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," vol. 49, pp. 87–100, 01 2014.

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

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

[15] A. Charguéraud, A. Schmitt, and T. Wood, "Jsexplain: A double debugger for javascript," pp. 691–699, 04 2018.

[16] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for JavaScript,” in *Proc. 16th International Static Analysis Symposium (SAS)*, vol. 5673 of *LNCS*, Springer-Verlag, August 2009.

[17] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, “Fast and precise type checking for javascript,” *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 48:1–48:30, 08 2017.

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

[19] D. Jang and K.-M. Choe, “Points-to analysis for javascript,” pp. 1930–1937, 01 2009.

[20] P. Gardner, S. Maffeis, and G. Smith, “Towards a program logic for javascript,” vol. 47, pp. 31–44, 01 2012.

[21] S. Maffeis and A. Taly, “Language-based isolation of untrusted javascript,” in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pp. 77–91, 2009.

[22] S. Maffeis, J. C. Mitchell, and A. Taly, “Isolating javascript with filters, rewriting, and wrappers,” in *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pp. 505–522, 2009.

[23] “Racket - general-purpose programming language.” Accessed on 2020-06-07.

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

[25] A. Charguéraud, “Pretty-big-step semantics,” in *Programming Languages and Systems* (M. Felleisen and P. Gardner, eds.), (Berlin, Heidelberg), pp. 41–60, Springer Berlin Heidelberg, 2013.

[26] “Coq - interactive formal proof management system.” `https://coq.inria.fr`. Accessed on 2020-06-07.

[27] 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.

[28] “V8 - google’s open source high-performance javascript and webassembly engine, written in c++.” `https://v8.dev`. Accessed on 2020-06-07.

[29] “K - rewrite-based executable semantic framework.” `http://www.kframework.org/index.php/Main_Page`. Accessed on 2020-06-07.

[30] 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.

[31] 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.

[32] G. Sampaio, J. F. Santos, P. Maksimović, and P. Gardner, “A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications,” in *34th European Conference on Object-Oriented Programming (ECOOP 2020)* (R. Hirschfeld and T. Pape, eds.), vol. 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 28:1–28:29, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[33] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, “Jsai: A static analysis platform for javascript,” FSE 2014, (New York, NY, USA), p. 121–132, Association for Computing Machinery, 2014.

[34] 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.

[35] E. Andreasen and A. Møller, “Determinacy in static analysis for jquery,” in *In Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, 2014.

[36] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, “Correlation tracking for points-to analysis of javascript,” in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP’12, (Berlin, Heidelberg), p. 435–458, Springer-Verlag, 2012.

[37] B. Livshits, “Jsir, an intermediate representation for javascript analysis.” `http://too4words.github.io/jsir/`, 2014. TypeScript language specification. Technical Report. Microsoft.

[38] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, “Javert 2.0: Compositional symbolic execution for javascript,” *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.

[39] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, “Symbolic execution for javascript,” PPDP ’18, (New York, NY, USA), Association for Computing Machinery, 2018.

[40] “Ocaml - general-purpose, multi-paradigm programming language.” `https://ocaml.org/`. Accessed on 2020-06-07.

[41] “Estree specification.” `https://github.com/estree/estree`. Accessed on 2021-07-30.

[42] “Esprima - a high performance, standard-compliant ecmascript parser written in ecmascript.” `https://esprima.org/`. Accessed on 2021-07-30.

[43] A. H. Gomaa, Wael ; A. Fahmy, “A survey of text similarity approaches,” vol. 68, April 2013.

[44] “Htmlsimilarity - compare html similarity using structural and style metrics.” `https://github.com/matiskay/html-similarity`. Accessed on 2021-07-27.

[45] G. Navarro, “A guided tour to approximate string matching,” *ACM Comput. Surv.*, vol. 33, p. 31–88, Mar. 2001.

[46] J. W. Ratcliff and D. E. Metzener, “Pattern matching: The gestalt approach,” *Dr. Dobb’s Journal*, p. 46, July 1988.

[47] P. Jaccard, “Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines,” *Bulletin de la Société Vaudoise des Sciences Naturelles*, pp. 241–272, 1901.