ECMARef6: A Reference Interpreter for Modern JavaScript

Rafael Rosa Rahal

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. António Paulo Teles de Menezes Correia Leitão
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Members of the Committee: Prof. Pedro Miguel dos Santos Alves Madeira Adão

May 2023

Acknowledgments

Abstract

ECMAScript is a highly influential modern programming language widely utilized in various environments, including server-side web applications, mobile development, and particularly client-side web applications. Despite its widespread popularity, the ECMAScript language specification is extensive and complex, posing challenges for typical developers to comprehend. To address this, we present ECMARef6, a reference interpreter designed for ECMAScript 6, the most popular version of the language. Our implementation faithfully follows the standard line-by-line, and undergoes rigorous testing against Test262, the official conformance test suite for ECMAScript. ECMARef6 represents the most comprehensive academic interpreter targeting ECMAScript 6, surpassing all other academic reference implementations in terms of test coverage. This project not only provides a robust implementation for ECMAScript 6 but also lays the groundwork for the implementation of the more recent versions of the ECMAScript standard.

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

Contents

List of Figures

Chapter 1

Introduction

JavaScript, also known as ECMAScript, is one of the most widely-used programming languages in the world. It has become the industry standard for dynamic front-end web development, and it is also utilized for server-side web applications using Node.js [1]. Additionally, it can run on a variety of devices including mobile and embedded systems with JerryScript [2]. JavaScript is not only the most popular programming language for the web, but it is also the most active language on GitHub,[1] and the second most active language on Stack Overflow.[2]

The ECMAScript Standard is the official specification for the language. Developed and maintained by the technical committee TC39, which is composed of language experts, browser vendors, and other stakeholders in the JavaScript community. The standard defines the syntax, semantics, and behavior of the language. The specification is a highly complex document written in the form of an interpreter with a mixture of pseudo-code and written text to explain its internal functionality.

ECMAScript is an ever-evolving programming language, which means that its standard must also keep pace with these changes. Currently, the Technical Committee 39 (TC39) releases a new version of the language every year. The evolution of the total number of pages in the ECMAScript standard is depicted in Figure 1.1. It is worth noting that the transition from ES5 to ES6 marked a significant milestone for the language, as evidenced by the substantial increase in the number of pages. This leap represented the most significant advancement in the language's development so far. Despite so many versions, the process of introducing new features is a significant manual undertaking. This process involves creating proposals, refining them through committee discussions, implementing them across various engines, and testing and documenting them. Unfortunately, this process is not only time-consuming, but it is also error-prone. Such manual steps can result in delays in adding new features to the language, thereby impeding its progress.

Introducing a reference interpreter for ECMAScript would provide several benefits and help control the language's evolution. A reference interpreter is a software implementation that conforms fully to the language specification. With such an interpreter in place, various issues can be addressed, including:

- **Backward compatibility**: The reference interpreter plays a crucial role in ensuring backward compatibility when making changes to the language specification. By running old test suites in the reference interpreter, it verifies that modifications to the language do not break existing code. This approach allows for the addition of new features while maintaining compatibility with previous code, preventing errors and inconsistencies.

- **Thorough testing**: The reference interpreter facilitates comprehensive testing of the language

[1] Github most active programming languages based on pull requests - https://madnight.github.io/githut/
[2] Stack Overflow Trends over time based on use of their tags - https://insights.stackoverflow.com/trends

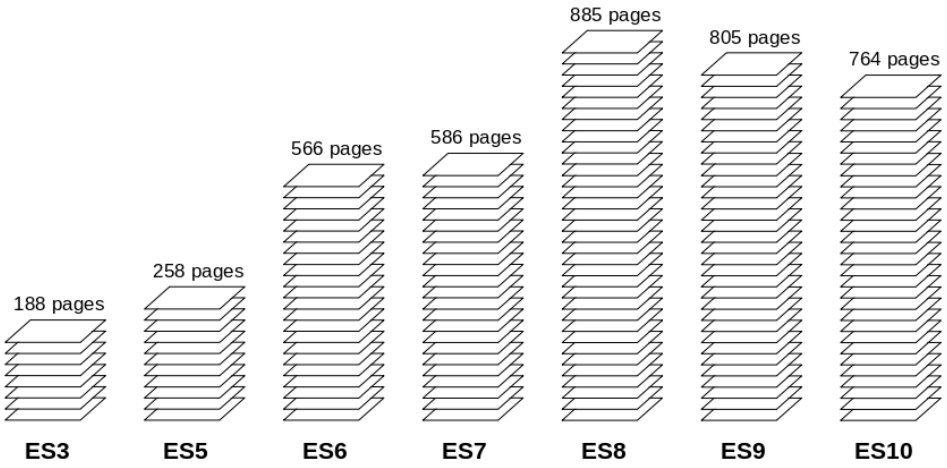


Figure 1.1: The evolution on the number of pages of the ES standard official document.

standard, aiding in the detection of specification bugs. By measuring test coverage and aiming to test the entire interpreter, it ensures that the language specification is consistent and free from ambiguities that could result in varying interpretations by different implementations.

- **Standardization of new features**: The reference interpreter greatly assists in the standardization of new features. It provides a clear specification for these additions, enabling developers to implement them efficiently and ensure compliance with the language standard. This helps prevent fragmentation and compatibility issues that may arise from the proliferation of non-standard language features. Additionally, the TC39 can leverage the reference interpreter to directly write and test code, which can then be used to generate complex HTML, streamlining the development process.

Overall, the introduction of a reference interpreter for ECMAScript would provide a standardized and consistent basis for the language's evolution, enabling the addition of new features while maintaining backward compatibility and ensuring a better programming experience for ECMAScript users.

In academia, numerous attempts have been made to develop a reference interpreter for ECMAScript. However, most of these reference interpreters only support version 5 of the ECMAScript standard [3; 4; 5; 6; 7]. There is only one project that targets a version beyond 5, which is documented in [8]. Nevertheless, this implementation still lacks comprehensive coverage of the entire language, particularly in terms of support for language built-ins. Consequently, this project only encompasses 18,064 tests out of a possible 35,990 tests included in the language official test suite [9].

To address this issue effectively, the ECMA-SL project was developed, consisting of two key components: a domain-specific language (DSL) specifically designed for standard specification and a reference interpreter built using this DSL. Figure 1.2 depicts the systems involved in the ECMA-SL project, emphasizing the language (ECMA-SL) and the existing interpreter (ECMARef5). However, the current interpreter has been limited to supporting only version 5 until now. The objective of this thesis is to enhance the existing interpreter, which already supports version 5, to also include support for version 6.

The transition from ES5 (version 5) to ES6 (version 6) marked a significant milestone in the evolution of the JavaScript language. ES6 introduced a plethora of new features such as arrow functions, template literals, destructuring assignments, and let and const keywords, which greatly improved the language's readability and ease of use. However, the changes did not stop at just adding new features. The internal structure of the document underwent a major overhaul, resulting in a complete refactor. This refactor was so comprehensive that to this day, the internal structure and algorithms have remained

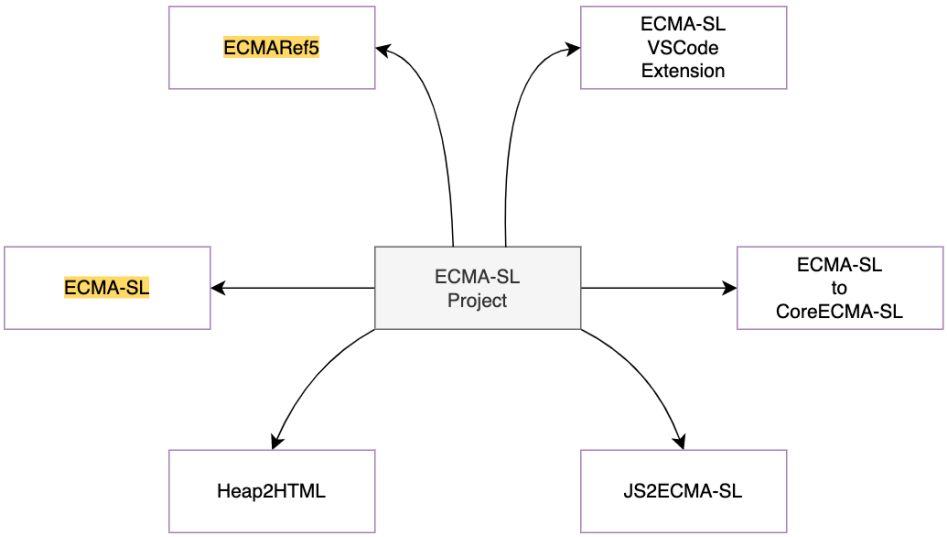


Figure 1.2: Systems in the ECMA-SL Project

almost unchanged, only being extended over time. Consequently, the first step towards supporting modern JavaScript is to have a functional version of the interpreter that can handle ES6. However, implementing an interpreter is an arduous task and is not feasible for a single thesis. Therefore, the task was divided into two theses, with the current one focusing on the core of the language, and the other one concentrating on the built-ins of the language.

Developing a reference interpreter for a complex real-world programming language, such as JavaScript, encompasses more than just an engineering task due to the typically incomplete nature of language specifications. Therefore, our role extended beyond merely translating the pseudo-code of the standard into our own DSL (Domain-Specific Language), as we also had to address the gaps in the specification. These gaps include defining suitable internal structures to represent the fundamental language concepts, striking the right balance between static and dynamic computation to ensure our interpreter aligns with the standard without sacrificing performance, and, perhaps most importantly, conducting an in-depth study of the extensive body of research on academic reference interpreters for JavaScript. By leveraging existing research on language specification, we were able to build upon prior work and enhance the quality of our interpreter.

As part of our development process, we placed a great emphasis on ensuring the quality and correctness of our work. To achieve this, we rigorously tested our interpreter using test262, which is the official JavaScript test suite. This involved carefully selecting the appropriate tests and making sure that our interpreter could successfully execute all of them. In the end, we were able to pass 92% of the selected tests, which is a testament to the reliability and robustness of our implementation.

This thesis is structured as follows. Firstly, in Chapter 2, we provide an in-depth analysis of the standard and ES6 language, highlighting key features such as objects, functions, and changes from ES5 to ES6. Next, in Chapter 3, we discuss a collection of related projects and papers in connection to the research conducted in this thesis. In Chapter 4, we present our solution, concentrating on the significant modifications and most critical work accomplished. Additionally, in Chapter 5, we describe our evaluation pipeline and results. Lastly, we conclude this thesis in Chapter 6.

Chapter 2

Background

2.1 ECMAScript

The ECMAScript Standard is the language specification for ECMAScript, which is the official name for the programming language commonly known as JavaScript. The language is maintained by TC39, a committee formed by developers, implementers and academics to maintain and evolve JavaScript. The group has regular meetings attended both by members and also by invited specialists.

In 2015, the TC39 committee changed the ECMAScript release process. Instead of trying to group updates to the standard into large batches of changes, the committee started to favor a more incremental approach, releasing a new version of the standard every year. This yearly update contains only the changes approved by the committee within that year. In fact, the 6th version of the standard was the last one under the old release process.

Because of this change, ES6 introduced a lot of user requested features, like arrow functions and template strings but also was the last major refactoring of the ECMAScript standard, effectively doubling the size of the standard from the previous version of the language, from 258 pages to 566 pages. But before diving into the changes, it is important to understand the high-level structure of the standard and how the JavaScript language works internally and is specified.

JavaScript has seven types of values on the language: `Undefined`, `Null`, `Boolean`, `String`, `Symbol`, `Number`, and `Object`. The latter is arguably the most important one because most of ECMAScript's features are based on objects, which are defined as collection of key-value pairs with the keys being of type `String` or `Symbol`. We refer to the keys of an object as its properties. Another important feature of objects is that they have prototypal inheritance, meaning that each object can be connected to a chain of object prototypes and inherit their properties.

The specification of the language is large and complex, being organized into 26 sections. This sections can be grouped in three main classes:

- **Syntax**: Corresponding to the definition of the lexical grammar of the language.
- **Core**: An aggregate of all the basic building blocks of the language such as expressions, statements and also the implementation of internal algorithms that will be fundamental for ES6 features, such as `ordinary objects` and `exotic objects`. This is the main focus of this project and will be further explained later on the report.
- **Built-ins**: All the libraries and APIs that are exposed by JavaScript programs and natively implemented by the JavaScript runtime using the features defined in the core; these include, for instance, the `String`, `Boolean` and `Number` built-in objects.

Figure 2.1 shows our division of the ECMAScript 6 standard as detailed above. The red rectangle represents the syntax part, which covers sections 10 and 11. The yellow rectangle represents the core, which goes roughly from section 7 to 16. At last, the blue rectangle represents the built-in objects that goes from section 18 to section 24.

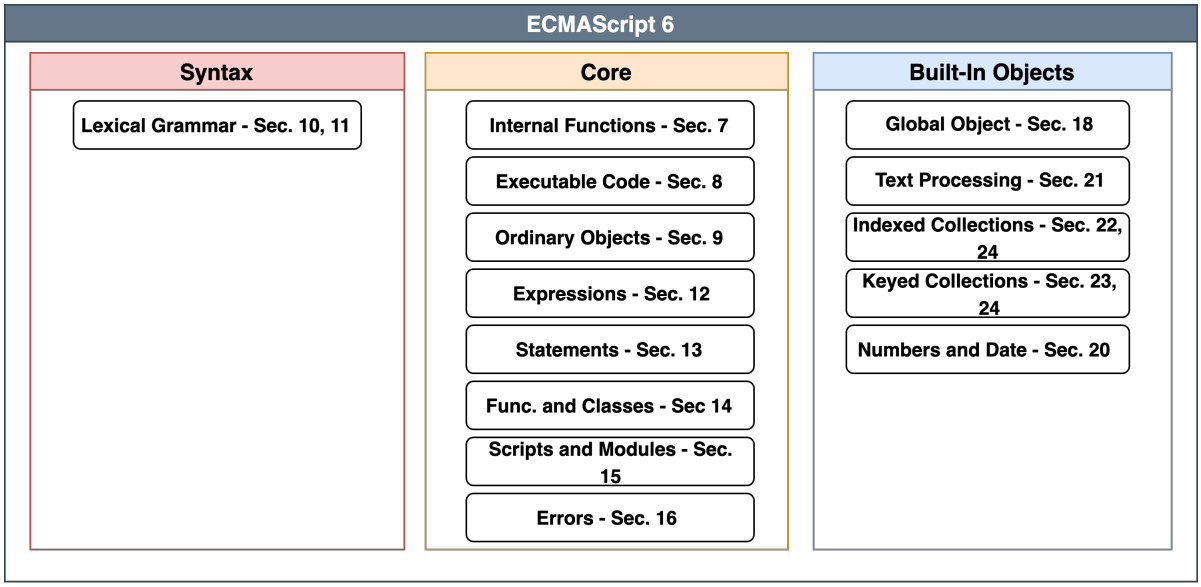


Figure 2.1: Our internal division of the ES6 Standard

2.2 ECMA-SL Project

The goal of the ECMA-SL project is to build an executable specification of the ECMAScript standard. To this end, a research team at IST developed a new language, called ECMA-SL, for the specification and analysis of the ES Standard, using which they developed ECMARef 5 [10; 11; 12], currently the most complete academic reference interpreter of ES5.

Figure 2.2 describes the execution pipeline of a given JavaScript program in the ECMA-SL project, which can be explained roughly in 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, a simplified version of ECMA-SL, obtaining the file `core.cesl`.
3. Interpreting the obtained Core ECMA-SL program using our ECMA-SL interpreter

In order to detail these steps, lets show how each one of the three components involved in this process work:

JS2ECMA-SL Given a file containing an ECMAScript program, the first step towards execution is the compilation using JS2ECMA-SL. This tool parses the program by using Esprima,[1] which is a standard-compliant ECMAScript parser. The output of Esprima is the AST of the program and this tree is then transformed into an ECMA-SL program that recreates it in ECMA-SL. To replicate the behavior of the

[1] Esprima - ecmascript parsing infrastructure for multipurpose analysis - https://esprima.org/

original JavaScript program we still need to call the ECMAScript interpreter on the ECMA-SL program. To accomplish this, we generate the program `out.esl`, which imports both the ES interpreter, `ESL_Interpreter.esl`, and the program for the AST, `ast.esl` and then we call the interpreter on the result of the combination as shown below:

```
import ast.esl
import ESL_Interpreter.esl

function main() {
      x := buildAST();
      y := JS_Interpreter_Program(x);
}
```

Listing 1: The entry-point for JS programs execution (`main.esl`)

Listing 1 shows the `main.esl` file, the entry-point for the program execution. This file imports the module `ast.esl`, which is the output of JS2ECMA-SL. It also imports the module `ESL_Interpreter.esl`, corresponding to our JS reference interpreter. The main function of this program simply constructs the AST of the original JS program in memory (via the `buildAST` function) and interprets the program by calling the function `JS_Interpreter_Program` with that AST.

ECMA-SL-to-Core The ECMA-SL program is then compiled to Core ECMA-SL, generating the program `core.cesl`. During the compilation to Core ECMA-SL all the imports included in the output of J2ECMA-SL are resolved. This means that the resulting program, `core.cesl` is self-contained and it includes all the code both from ECMARef as well as the code of the program to be run.

ECMA-SL interpreter The resulting Core ECMA-SL program is then interpreted using the ECMA-SL interpreter, which was written in OCaml [13]. There are two main modes for the interpretation: silent and verbose. When in silent mode, the interpreter just outputs the final ECMA-SL heap generated by the execution of the program. In verbose mode, the interpreter also logs the sequence of executed commands for debugging purposes.

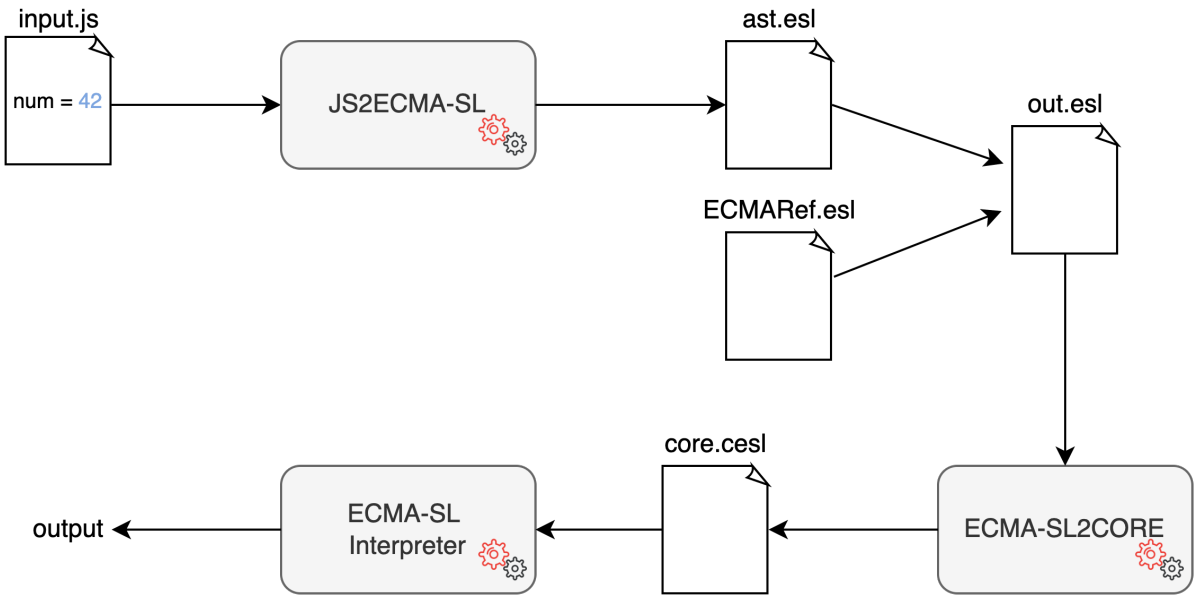


Figure 2.2: Execution Pipeline

2.3 ES6 Core

One can divide the ES6 language into three main components: syntax, core and built-in objects. We use the term ES6 core to refer to Sections 7 to 16 of the ES6 standard. These sections describe the main building blocks of the language.

The following subsections give a detailed description of the ES6 language based on the standard. We put the focus on the core components of the language, which includes: objects, functions, scope and the semantics of expressions and statements.

2.3.1 Objects and Properties

The ES6 language is object-based, which means that basic language and host facilities are provided by objects and consequently an ES6 program is a cluster of communicating objects. Ordinary Objects are the most common form of objects and they are defined as any object with the default semantics presented in section 9.1 of the standard. In opposition to that, the standard defines *Exotic Objects* as any object that is not an *Ordinary Object*. Both of these concepts were only introduced in the 6th version of the standard.

Any object, either ordinary or exotic, is defined as a collection of properties. These properties are containers that hold other objects, primitive types or functions. Apart from its value, a property also contains attributes that determine how it is used. Each object property can be described as one of these three classes: an internal slot, a named property, or a named accessor property. It is possible to add/remove properties to/from objects during execution, which means that objects are dynamic.

Internal slots vs Named Properties Internal slots are properties that provide meta-information about the object they are associated with, for instance, the object above in the prototype chain. Most of these properties are used on the implementation of internal algorithms and operations presented in the standard. These properties are not inherited by the prototype chain and cannot be accessed directly by an ES6 program. Every ordinary object contains at least these two slots:

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

Properties explicitly created by the program are called named properties, which are divided into two subtypes: named data properties and named accessor properties. Their differences are discussed later in this section.

Listing 2 and Figure 2.3 illustrate the creation and representation of an object. The listing declares the variable `o` using the keyword `const` and assigns it to an object containing two properties: `answer` and `nested`. The object resulting from the evaluation of this assignment is shown in Figure 2.3. The two named properties are represented as blue rectangles and the two default internal slots as red rectangles. Note that each blue rectangle points to a yellow rectangle, which represents a property descriptor (discussed in the subsection below) containing the value of the respective property and meta-information about the property itself.

Property Descriptors When explicitly set by the program a property is called a named property, this can be done either through built-in functions or assignment expressions. Named properties are repre-

```
const o = {
    answer: 42,
    nested: {
        hello: "world"
    }
}
```

Listing 2: An object declaration in ES6

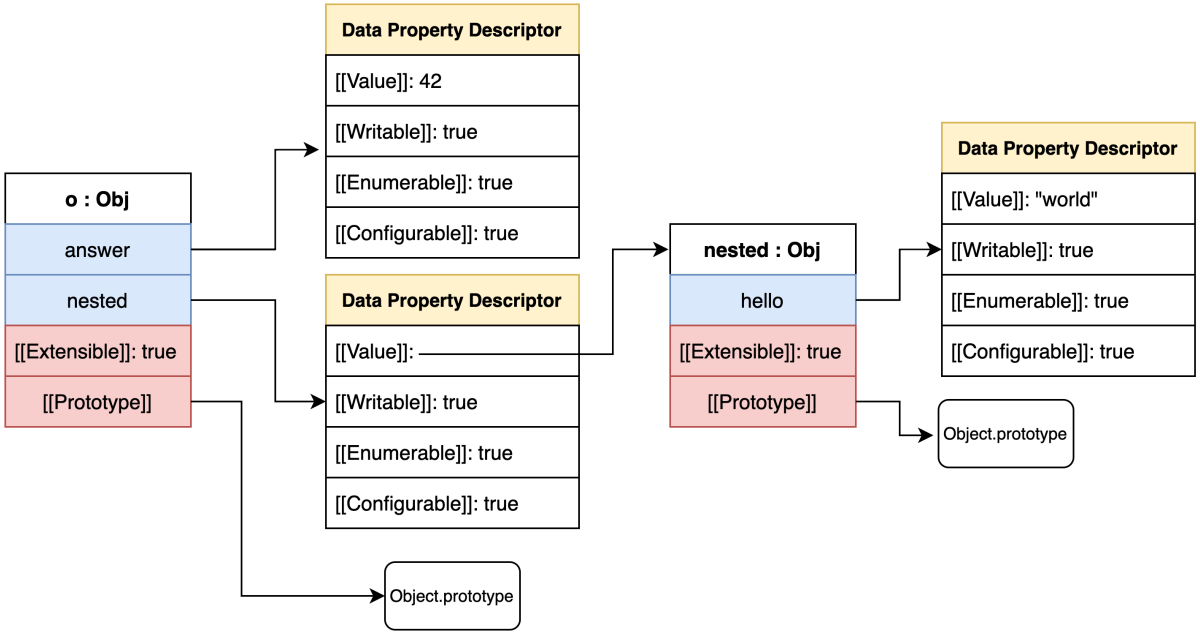


Figure 2.3: Object representation of Listing 1

sented by *property descriptors*. Those descriptors are records with specific attributes describing 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 illustrating with an example the most important one.

Data Property Descriptors A data property descriptor stores meta-information about the property itself and also holds the value of that property. Each descriptor contains a record with four attributes:

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

4. **[[Configurable]]** a boolean that 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 ES6 data property descriptors, described in the yellow rectangles. Note that the value of the property `nested` is stored inside the [[Value]] attribute. This attribute points to another object, which also has its own named properties with their own descriptors.

In ES6, a program can update or create new properties in two ways: via property assignment expression or via a built-in function `Object.defineProperty`. This less usual way lets the program specify the attributes associated with the given property descriptor. The attributes that store meta-information on the property, [[Writable]], [[Enumerable]], and [[Configurable]] have different values depending on the way a property is defined. There are two possible scenarios:

1. When using the object literal notation, e.g. `{ foo: "banana" }`, all these attributes are true by default;
2. When using the built-in function `Object.defineProperty`, if any of the attributes 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:

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

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 ECMAScript program.

2.3.2 Function Objects

In ES6, a function object is an specialisation of the ordinary object. This means that functions also have the two default slots, [[Prototype]] and [[Extensible]]. Apart from those, the function object also has to store data that describes information about the function, such as: its parameters, the scope and the code of the function. Here are some of the additional slots that are found in function objects:

1. `[[Environment]]` stores the scope chain in which the function was created (discussed in Section 2.3.3);
2. `[[FormalParameters]]` contains the list of the function's formal parameters;
3. `[[ECMAScriptCode]]` contains the code of the body of the function;

4. `[[FunctionKind]]` contains the type of the function. A string with one of these three values: `"normal"`, `"classConstructor"`, or `"generator"`;

5. `[[ThisMode]]` defines how the `this` keyword will work inside the function. Either `lexical`, `strict`, or `global`.

Note that, function objects in ES5 only had three dedicated internal slots, the first three defined in the list above. The slots [[FunctionKind]] and [[ThisMode]] were only added in ES6, alongside with other slots omitted in this report. Beside internal properties (slots), all ECMAScript function objects also have internal methods which are responsible for operations related to functions, we choose to illustrate the `[[Call]]` method which is responsible for the execution of the function itself.

Figure 2.4 shows the specification of the method by the standard. It follows these steps: first the standard checks if the function can be executed (lines 1 and 2); then it stores both the caller context and callee context on variables (lines 3 and 4); after that the callee context becomes the running execution context and we make sure that the `this` keyword is referring to the correct object (lines 5 and 6); the execution of the code happens in line 7 and the restoration of the previous context in line 8; at last, a completion (discussed in Section 2.3.4) is returned based on the result of the code execution (lines 9 to 11).

9.2.1 [[Call]] (thisArgument, argumentsList)

The [[Call]] internal method for an ECMAScript function object *F* is called with parameters *thisArgument* and *argumentsList*, a List of ECMAScript language values. The following steps are taken:

1. Assert: *F* is an ECMAScript function object.
2. If *F*'s [[FunctionKind]] internal slot is **"classConstructor"**, throw a **TypeError** exception.
3. Let *callerContext* be the running execution context.
4. Let *calleeContext* be PrepareForOrdinaryCall(*F*, **undefined**).
5. Assert: *calleeContext* is now the running execution context.
6. Perform OrdinaryCallBindThis(*F*, *calleeContext*, *thisArgument*).
7. Let *result* be OrdinaryCallEvaluateBody(*F*, *argumentsList*).
8. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
9. If *result*.[[type]] is return, return NormalCompletion(*result*.[[value]]).
10. ReturnIfAbrupt(*result*).
11. Return NormalCompletion(**undefined**).

Figure 2.4: The [[Call]] internal method from Function Objects

2.3.3 Functions and Scope

In ES6, the standard uses execution contexts to track the runtime evaluation of code. There are three different types of execution context: global, function, and eval code. Global code is the code that is at the top-level of an ES6 program and not inside functions. Function code is the code corresponding to the body of a function. Finally, eval code is the code dynamically evaluated using the eval operator, e.g. `eval("x " + "= 2")`.

Each execution context contains a Lexical Environment. A Lexical Environment is used for storing the bindings of variables and functions identifiers and it is defined by the standard as a type with its own operations. Every Lexical Environment contains an Environment Record and a (possibly null) reference to an outer Lexical Environment. An Environment Record can be seen as a lookup table of

key-value pairs because it maps all the variables and parameters associated with a given scope to their respective values. Scope resolution is performed by inspecting the identifier bindings present in lexical environments.

Listing 3 represents a program that deals with different lexical environments. In lines 1 and 2 the program associates the global variables `x` and `y` to the values `1` and `"Hello"`. After that, a function named `func1` is created and inside that function we associate a variable `x` with the value `2`. The last part of the listing shows the declaration of the function `func2` inside `func1`, on this new function we have the definition of variable `x`, this time pointing to the value `3`.

Figure 2.5 shows the lexical environments created during the execution of Listing 3. The global lexical environment (blue rectangle) has its outer lexical environment pointing to null and its environment record contains: `x` and `y` with the values `1` and `"Hello"`; and `func1` storing a function object. The lexical environment of `func1` (yellow rectangle) has its outer lexical environment pointing to the global lexical environment and its environment record contains the `x` variable with the value `2` and `func2` with its own function object. At last, the lexical environment of `func2` has its outer lexical environment pointing to the one owned by `func1` and its environment record contains only the variable `x` associated with the value `3`.

```
1  let x = 1
2  let y = "Hello"
3  function func1() {
4      let x = 2
5      function func2() {
6          let x = 3
7      }
8  }
```

Listing 3: An example to illustrate the scope handling in ES6

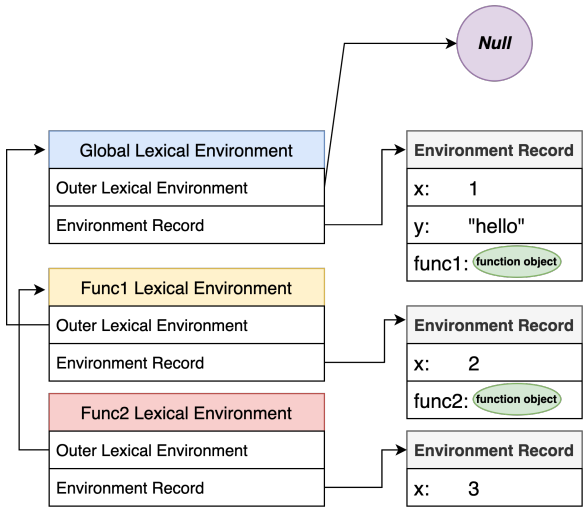


Figure 2.5: Scopes representation example of Listing 3

2.3.4 Expressions and Statements

The following subsections aim to illustrate the semantics of the various constructs of the language specified by the standard. To this end, we explore in detail the *If-Then-Else* statement and the equality

expression. Before diving into the specification of the semantics, we must introduce references and completions.

References A reference is a representation of an unresolved name binding and it is formed by two entities: a base value and a referenced name. The base value is either undefined, an object, a boolean, a string, a symbol, 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 either a string or a symbol and it is usually 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. 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 whose prototype chain that property can be found. For instance, the reference `l.p` denotes the property p in the prototype chain of the object at location l. To obtain the associated value, the reference needs to be dereferenced. This is the job of the internal function, defined in the standard, called `GetValue`.

Completions Completions are represented by records in the standard. A completion record contains meta-information about the evaluation of a statement; more precisely, a completion record includes the following fields:

- **[[type]]** representing if the evaluation of the given statement terminated with a value, a return statement, a break or a throw;
- **[[value]]** storing the resulting value of that operation (if any);
- **[[target]]** holding the target for when a flow transfer happens, e.g, when the type is return.

Any completion that has a `[[type]]` other than normal is referred to as an *abrupt completion*.

If-Then-Else In Figure 8, we show the specification of the If-Then-Else statement presented in the ES6 standard. This statement is described in 8 steps, on ES5 it was described in 3 steps. The specification goes as follows: it starts with the evaluation of the guard expression, returning the reference `exprRef` (line 1); then there is the conversion of this reference to a boolean using the internal functions `ToBoolean` and `GetValue` (line 3); and the evaluation of one of the two statements (then or else) depending on whether the value returned is true or false; the result is stored in the `stmtCompletion` variable (lines 4a and 5a). At last, we return the completion if its value is not empty or a normal completion with the value undefined. Note that, during the evaluation, the standard uses the `ReturnIfAbrupt` (lines 3 and 6) internal function in order to check if an exception was thrown to terminate the execution in that case.

Equality Operations In ES6, the basic equality operator is the double equal operator, `==`, which compares the value of its right operand to the value of its left operand. The standard also defines other equality and comparison operators, that are shorthand for other standard operations. For instance, the strict equality operator, `===`, a stricter way of comparing values, not performing implicit type coercions.

Figure 9 shows the semantics of the simple equality defined in the ES6 standard. Consider the following expression: `x == y`. We can describe the semantics of this expression as this: evaluate the left operand, assigning the returned reference to `lref` (line 1); get the value associated with `lref` with the `GetValue` internal function and assign it to `lval` (line 2); repeat the same steps with the right operand, generating `rref` and `rval` (lines 4 and 5); after that we return the result of the Abstract Equality Comparison algorithm described in the standard. The algorithm is called with the parameters `lval` and `rval`.

13.6.7 Runtime Semantics: Evaluation

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

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
3. ReturnIfAbrupt(*exprValue*).
4. If *exprValue* is **true**, then
    a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
5. Else,
    a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
6. ReturnIfAbrupt(*stmtCompletion*).
7. If *stmtCompletion*.[[value]] is not empty, return *stmtCompletion*.
8. Return NormalCompletion(**undefined**).

Figure 2.6: The `If-Then-Else` specification

12.10.3 Runtime Semantics: Evaluation

EqualityExpression : *EqualityExpression* `==` *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of performing Abstract Equality Comparison *rval* == *lval*.

Figure 2.7: The equality (==) operation specification

2.4 From ES5 to ES6

In ES6, the ECMAScript standard introduced key differences when compared to the previous version. These changes range from the addition of new features to refactoring of the standard's internal algorithms. The following subsections describes the changes related to the core of the ES6 language, namely: the get and set functions, treatment of exceptions and the built-in objects.

2.4.1 The Get and Set Functions

The `Get` and `Set` functions are key factors on the language because they are the interface responsible for retrieving and placing values on objects. These operations are crucial for the overall behavior of the language. Therefore, we detail here the changes for each of those two functions.

Get ES5 had the `[[Get]]` function implemented as a method common for all objects. This method would try to access some property `P` associated with that object and return `undefined` if the property was not found. On the new version, the `[[Get]]` method belongs to every ordinary object. The signature of the function changed as it now receives the property `P` and the original object that requested the

property, called `Receiver`. Another difference is that this new method searches for the property not only inside the object but also in the prototype chain, on the old implementation this task was performed by the `[[GetProperty]]` method. Figure 2.8 compares the two different implementations and highlights the new search in the prototype chain, now inside the method itself.

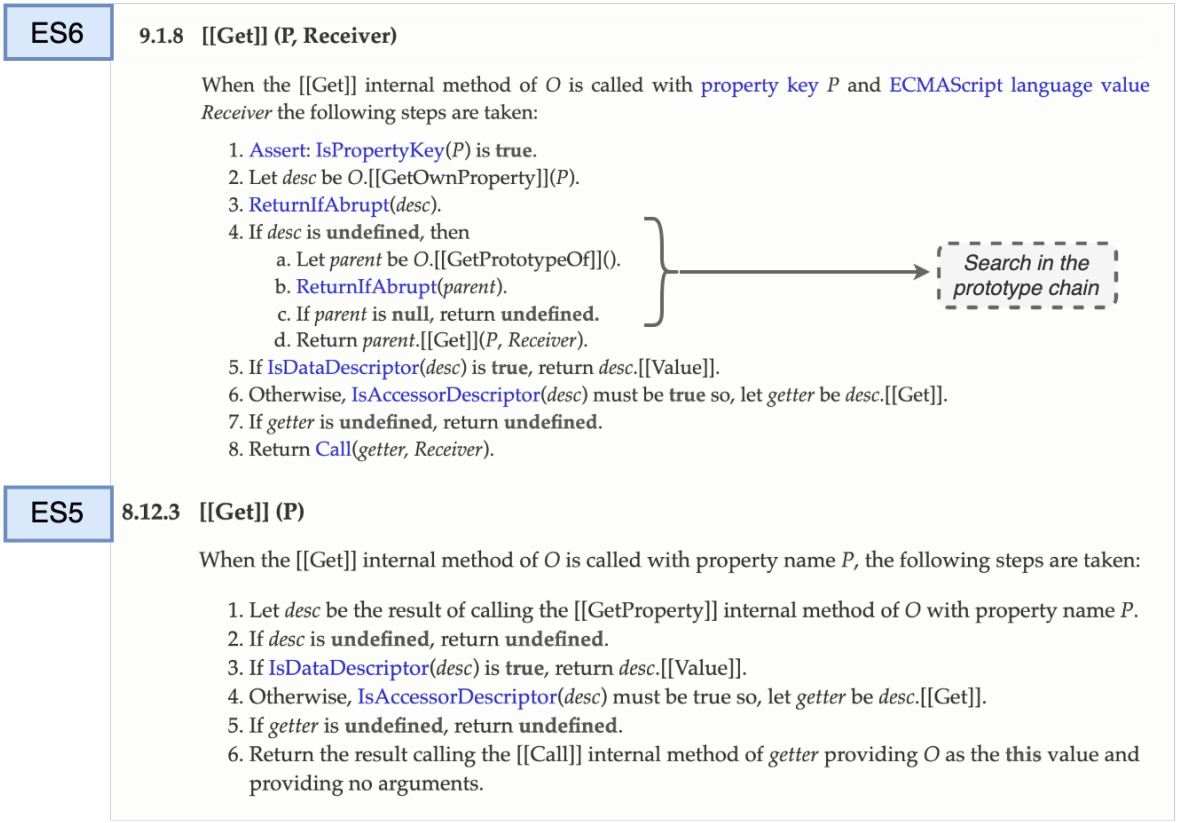


Figure 2.8: The different implementations of the `[[Get]]` method

Set On the previous version the main method to set attributes inside an object was the `[[Put]]` method, also attached to the object implementation. This function would receive a property `P`, a value `V` and a flag `Throw` indicating if the method should throw an exception or not. In ES6, the method for inserting properties is called `[[Set]]` and it is also attached to the ordinary object implementation. The new method receives a property `P`, a value `V` and the original object that requested the operation, called `Receiver`. This means that the [[Set]] method does not throw an exception anymore, it simply returns true when it is able to set the property and false when it is not.

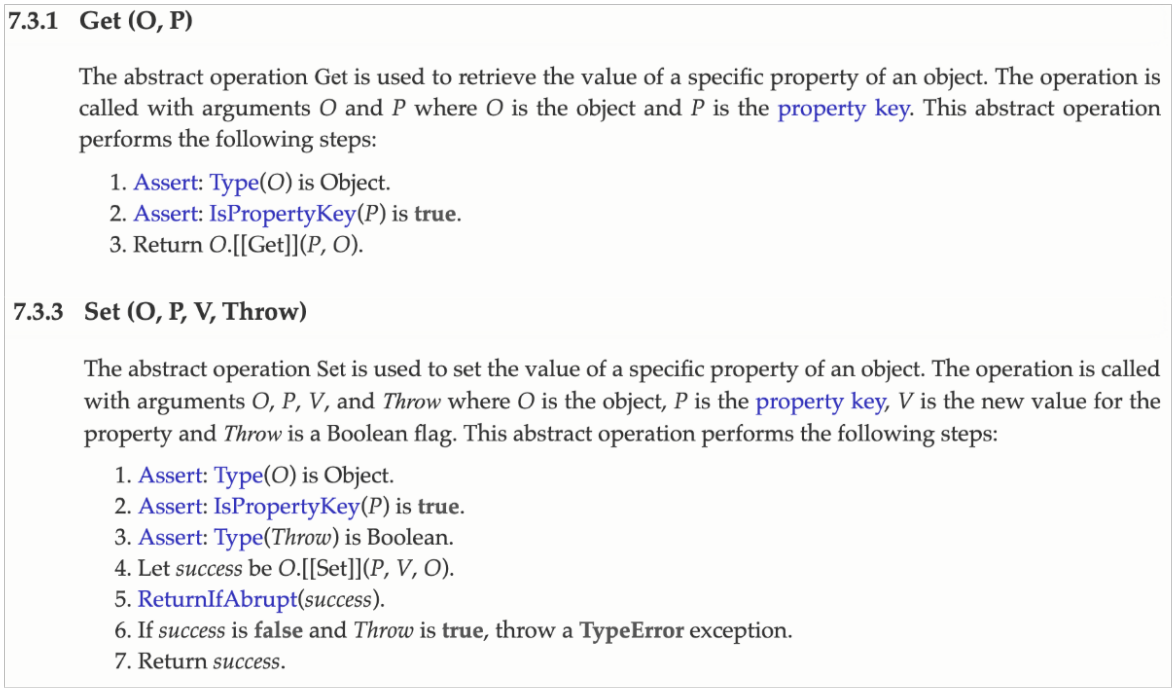
Apart from this changes inside the methods belonging to objects, the standard also added internal functions with similar purposes. The `Get` and `Set` internal functions are not attached to any implementation. They act as a safer way of getting and setting properties.

In the case of the `Get` function, its first responsibility is to type check the property and the object that made the request. After that, it calls the `[[Get]]` internal method to execute the operation.

The `Set` function also checks the type for the object and the property. Besides that, it also receives a `Throw` flag and throws an exception when the flag is true and it was not able to set the property. The execution of the operation is also delegated to the internal method `[[Set]]`.

Figure 2.9 shows the specification of these two internal functions. The first lines on each function type checks the parameters. After that, these functions delegate the operation to the methods tied to

the object. It becomes clear then, that despite having similar names, these functions have different responsibilities.



7.3.1 Get (O, P)

The abstract operation Get is used to retrieve the value of a specific property of an object. The operation is called with arguments *O* and *P* where *O* is the object and *P* is the property key. This abstract operation performs the following steps:

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Return *O*.[[Get]](*P*, *O*).

7.3.3 Set (O, P, V, Throw)

The abstract operation Set is used to set the value of a specific property of an object. The operation is called with arguments *O*, *P*, *V*, and *Throw* where *O* is the object, *P* is the property key, *V* is the new value for the property and *Throw* is a Boolean flag. This abstract operation performs the following steps:

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Assert: Type(*Throw*) is Boolean.
4. Let *success* be *O*.[[Set]](*P*, *V*, *O*).
5. ReturnIfAbrupt(*success*).
6. If *success* is **false** and *Throw* is **true**, throw a **TypeError** exception.
7. Return *success*.

Figure 2.9: The Get and Set internal functions of ES6

2.4.2 Treatment of Exceptions

In ES5, the exception handling was made by internal algorithms that would throw an exception depending on the execution. Despite throwing an exception, the execution would still run because the completion generated by the throw would only be captured at statement level. ES6 changed that by introducing changes that will deal with exceptions in a different way, namely:

1. Some functions now have more explicit names to be clear that they throw exceptions, for example we have *CreateDataPropertyOrThrow* and *DeletePropertyOrThrow* methods.
2. A new internal function called **ReturnIfAbrupt** was added. This internal function is responsible for checking if the argument is an *Abrupt Completion* and if it is, it returns immediately without the need to execute the rest of the function.

This new operation is included in many internal algorithms of the language. There are more than 1000 references to this function call on the ES6 standard page. Figure 2.10 presents the `GetValue` function (already discussed in Section 2.3.4) both on ES5 and ES6. The specification is almost identical, the only difference is that on ES6 (the top definition) we have a `ReturnIfAbrupt` call as the first step of the execution.

2.4.3 The Built-in Objects

Built-in objects are the structures that are available to the developer when using ECMAScript. They include the *Global Object*, which is part of the global lexical environment and stores the global state, `numbers`, `strings`, `booleans` and all other values and structures of the language.

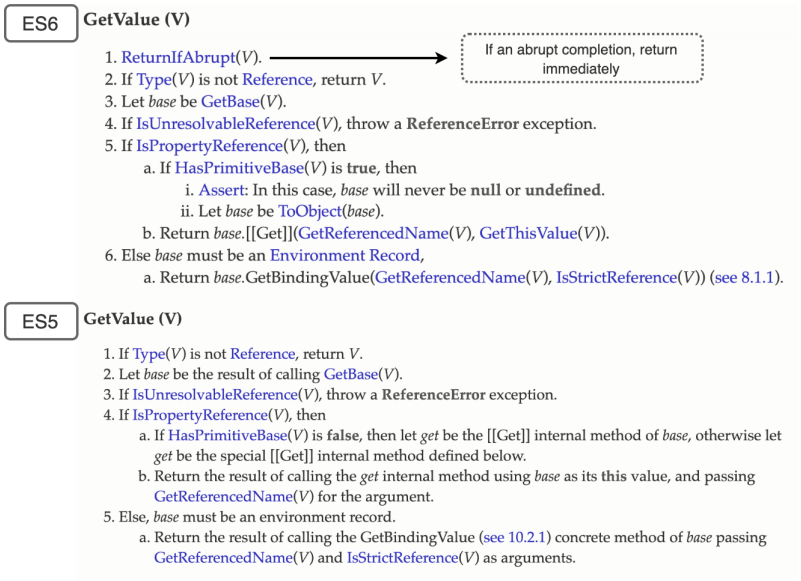


Figure 2.10: The `GetValue` specification both for ES5 and ES6.

In ES6, the built-ins were extended to support new features. To this end, the standard either created new built-ins libraries or expanded the existing ones. All of the built-ins are constructed on top of concepts presented above, like ordinary objects, function objects and others.

Figure 2.11 illustrates all built-ins defined in ES6. The yellow rectangles represent libraries introduced in previous versions of the standard. The red rectangles represent new libraries introduced in the ES6 version.

As one can notice from the image, there is much work around the built-ins. Some of it is adapting the ES5 implementation to ES6 and some of it is writing new libraries from scratch. Because of that, we will have the help of intern students to advance on the work of other libraries that are further away from the core.

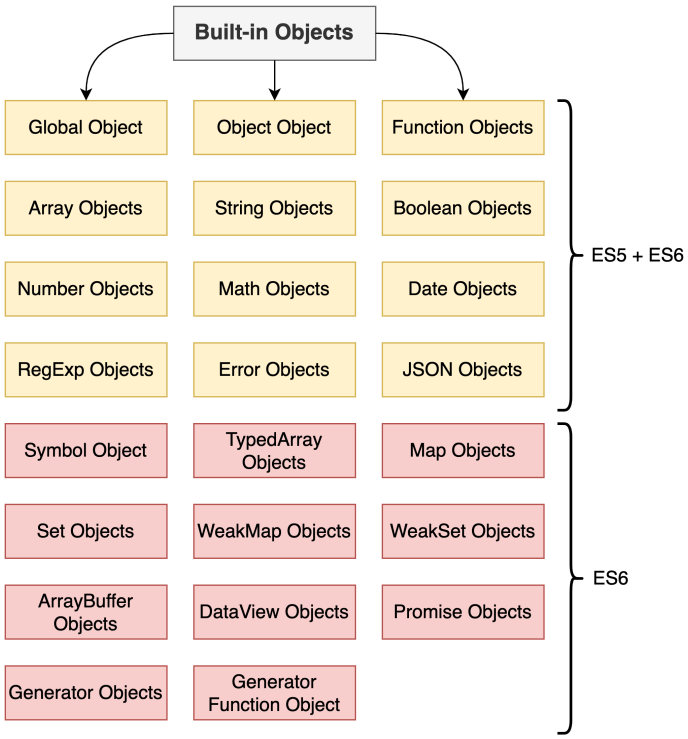


Figure 2.11: A scheme showing the built-ins in ES5 and ES6

Chapter 3

Related Work

The research literature includes a variety of topics on program analysis techniques for JavaScript, including: type systems [14; 15], points-to analyses [16], control-flow analyses [17], abstract interpretation [18; 19], information-flow analyses [20; 21; 22], and program logics [23; 7; 24]. On this section, we give an overview of the work that tries to formalise ECMAScript, including reference implementations of the language.

The originality of our ES6 reference interpreter is that it is designed to follow line-by-line the ECMAScript standard. In contrast, existing reference interpreters/formalisations are often substantially different from the text of the standard. This difference results in two drawbacks: first, most of these interpreters/formalisations are written in highly technical/mathematical formalisms (such as Coq [25] and K [26]) which are difficult to understand by nonacademic programmers, as the majority of the ECMAScript committee; second, the level of trust on these reference interpreters relies heavily on testing against Test262, which is known to have severe coverage issues. The first problem is particularly important considering that our goal is to argue for an executable JavaScript specification instead of the current textual one. The effort of moving from the textual description would be considerably smaller when using an interpreter written in the ECMA-SL language.

In the following, we give a brief summary of the most relevant research papers that present formalisations of the JS standard.

The first formal operational semantics of JavaScript Maffeis et al. were the first to design an operational semantics for JavaScript [27]. 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 [28; 29]. This semantics was the first to follow the standard formally, covering most of its internal functions and behaviours.

Lambda calculi for reasoning about JavaScript code: λS and S5 λJS is a core lambda calculus developed by Guha et al. [30] which supports features that are critical for ES3 such as: extensible objects, prototype-based inheritance, and dynamic function calls. They used `Racket` [31] to write their interpreter for λJS and also developed a compiler from ES3 to λJS. Alongside with that, they added a type system for checking a simple confinement property of λJS expressions. On the other hand, the authors do not test their compiler against Test262 and only support a subset of ES3.

Politz et al. further improved λJS with the addition of ES5 support, namely for property descriptors, getters, setters and the eval statement [32]. This version was called S5 and was also implemented in `Racket`. In contrast to λJS, S5 was thoroughly tested against Test262, passing 8,157 tests out of a total of 11,606 ($\approx$70%). Despite the good results, the authors claim that they only implemented 60% of the

ES built-in objects, those missing objects are the ones responsible for the failing tests. Furthermore, they also have tests related to non-strict code that are failing, showing that S5 is not entirely consistent with the ECMAScript standard.

Mechanised semantics of JavaScript: JSCert and KJS Bodin et al. [4] developed a formalisation of the semantics of ES5 written in `Coq` called JSCert. The authors adopted a pretty-big-step semantics [33] of ECMAScript 5. Other than an operational semantics, they also implemented a reference interpreter, JSRef, which they prove correct with respect to the defined operational semantics and tested it against a fragment of Test262. They were able to execute the tests by using `Coq-to-OCaml` extraction mechanism, generating an `OCaml` version of the JSRef interpreter. JSRef passes 1,796 tests out of a total of 2,782 tests corresponding to the core features of the language, as it does not support most of the ES5 built-in objects. These failing tests are the ones that make use of built-in objects which were not implemented, one example would be the ES array.

Gardner et al. [5] continued the work on JSRef by adding the support for ES5 Arrays by linking it to Google's V8 [34] Array built-in object implementation. In this second paper, the authors included a breakdown of all passing and failing tests, alongside with a detailed account of the testing infrastructure used to evaluate the JSCert project. More concretely, with the inclusion of the V8 Array built-in object, JSRef passes 2,440 core language tests out of 2,782, and 1,309 Array tests out of 2,267.

Park et al. presented KJS [6], a formal semantics of ES5 written in the K framework [26], which is a well established term-rewriting system supporting various types of symbolic analyses. This implementation was tested against the Test262 passing 2,782 core language tests. The K framework has a built-in symbolic analysis mechanism based on reachability logic [35]. Because of that, one can symbolically analyse JS programs by combining KJS with these symbolic facilities. The authors also demonstrated how this strategy can be used in practice by verifying various data structures and sorting algorithms implemented in ECMAScript [36], namely: AVL tree, binary search tree, quick sort, merge sort, among others.

JSExplain and JaVert Charguéraud et al. [3] presented a reference interpreter for ECMAScript 5 called JSExplain. This interpreter follows the text of the specification closely. It also allows programmers to code-step not only to their JS code but also the pseudo-code of the standard. JSExplain was written in OCaml, using a purely functional subset of the language extended with a built-in monadic operator for automatically threading the implicit state of the interpreter across pure computations [37]. Additionally, the authors also have a version of JSExplain that runs in the browser, by implementing a compiler from their purely functional subset of OCaml to JavaScript. The primary goal of JSExplain was to provide both the state of the program and the internal state of the ECMAScript interpreter, allowing programmers to debug the execution of JS code. Despite that, there are key differences from this work to JSExplain especially the fact that JSExplain was not thoroughly tested against Test262 and has limitied support for JS built-in libraries.

Apart from complete implementations of the standard, there were stand-alone reference implementations related to some of the built-in libraries inside the ECMAScript standard. Sampaio et al. developed a reference implementation of JavaScript Promises that follows the standard line-by line [7] with which the authors built a symbolic execution tool for analysing ECMAScript programs that use promises [38].

JISET Recently, J. Park et al. introduced JISET, an instruction set designed to be a compilation target for ES code. JISET focus on ECMAScript 10 with an extraction mechanism to semi-automatically create an ES to JISET compiler from the standard's text. The authors were able to test the execution engine

for JISET, passing 18,064 out of 35,990 available tests.

Despite the similarity between JISET and the ECMA-SL project, JISET does not support most of ES built-in libraries. Most complex object implementations such as RegExp, JSON and String were not implemented because they require advanced programming language techniques. In contrast with that, ECMARef5 supports all built-in objects.

ECMARef5 ECMARef 5, the reference interpreter developed in the context of the MSc theses of L. Loureiro [10], D. Gonçalves [11] and F. Quinaz [12]. This interpreter was thoroughly tested against Test262 and also supports all the built-in object of ES5. Consequently, ECMARef5 has 12,026 passing tests out of a total of 12,074 ES5 tests. Apart from the confidence on tests, this work also is line-by-line compliant with the ES5 standard. Furthermore, a tool called HTML2ECMA-SL was created in order to generate interpreter code from the text of the ECMAScript standard.

Summary As one can note, the work around implementations of the ECMAScript standard is extensive. Figure 3.1 compares the number of passing tests on some of those projects. It also shows in which language these projects were implemented. The image shows that ECMARef5 is the most complete academic reference implementation of the ECMAScript 5 standard, passing 12,026 tests out of 12,074 applicable tests. Despite passing 18,064 tests, JISET targets the 10th version of the ECMAScript standard. Because of that, it has a larger pool of available tests and the 18,064 tests correspond only to 62,4% of all the ES10 tests. Additionally, neither JISET nor any other implementation have support for all ES5 built-in objects, which ECMARef 5 does.

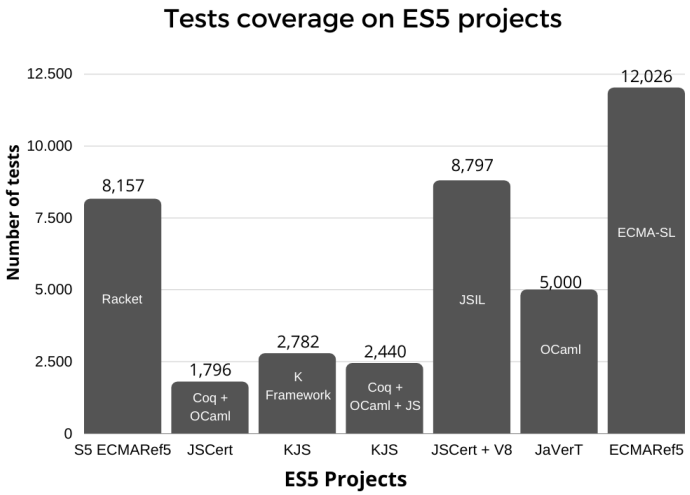


Figure 3.1: The test coverage on ES5 related projects and their used languages.

Chapter 4

ES6 Reference Interpreter

This chapter describes the implementation of ECMARef6, a reference interpreter for the ES6 version of the ECMAScript language. It is noteworthy that this language can be separated into two primary components, namely the core and the built-ins. This study mainly focuses on the development of the core aspect of the language, along with providing support towards the development of the built-ins. Nonetheless, the core aspect of the language demanded most of my attention, particularly regarding Functions, Scope, and Classes. I led the majority of the development of Functions and Classes, while I received valuable aid from my colleague Manuel Costa concerning the topic of Scope. This chapter offers readers a detailed account of these sections of the ECMAScript language and their implementation in ECMARef6.

4.1 Functions

The transition from ES5 to ES6 brought a significant refactoring to the internal representation of functions in the standard, along with newly added features. This refactoring included changes to various internal functions related to ECMAScript functions including those responsible for function creation, function execution, and context preparation. As a result, the entire function infrastructure of ECMARef had to be re-written while maintaining compatibility with ES5.

To gain a deeper understanding of functions within the standard, let us examine the lifecycle of a function in an ECMAScript program. Functions typically undergo two distinct stages: declaration and execution. During declaration, the function's abstract representation is stored in memory, while execution ensures the function is invoked within the correct context. The next subsections outline these two stages.

4.1.1 Function Declaration

In ECMAScript, there are two fundamental types of functions: named functions and anonymous functions. Named functions are explicitly declared with a specific name and can be invoked by that name throughout the code. Anonymous functions, on the other hand, do not have a name attribute and are created as a result of an expression. They can be defined using either the `function` keyword or using arrow functions, the new syntactic construct introduced in ES6. More specifically, arrow functions offer a more concise way of defining anonymous functions through the use of the => syntax.

Listing 4 compares three ways of defining functions in ECMAScript. The first function, `sum_one`, is a regular named function defined using the `function` keyword. The second function, `sum_two`, is an anonymous function also defined using the the `function` keyword. The third function, `sum_three`, is an anonymous function defined with the special arrow syntax. It is worth noting that in arrow functions, the

function block is optional; when it is omitted, the expression corresponding to the body is immediately returned. This is the case of the last function, `sum_four`.

```
function sum_one(a, b) {
        return a + b
}

var sum_two = function(a, b) {
  return a + b
}

var sum_three = (a,b) => {
  return a + b
}

var sum_four = (a,b) => a + b
```

Listing 4: Different ways of defining the same function in ES6

The functions defined in Listing 4 would generate three distinct node types in the program's Abstract Syntax Tree (note that `sum_three` and `sum_four` are represented by the same type of node). Let us briefly describe the three JavaScript AST nodes that represent function definitions:

- **FunctionDeclaration**: This AST node is used to represent a normal named function declaration, as seen in the first example in Listing 4. It is treated as a statement, i.e., a high-level instruction in the language;
- **FunctionExpression**: This AST node is used to represent an anonymous function expression declared using the function keyword, like the second declaration in Listing 4. It is treated as an expression and is typically used inside other statements, such as variable assignments;
- **ArrowFunctionExpression**: This AST node is used to represent an anonymous function defined using arrow functions, as in the third declaration in Listing 4. It is also an expression.

When interpreting a function definition, ECMARef6 creates a *Function Object* that represents that function in memory. If the function is named or assigned to a variable, as in Listing 4, the function object is associated with a binding in the current scope. The topic of scope will be explored in §**??**.

All types of functions in ES6 are internally represented as *Function Objects*, which means they also benefit from the prototype-based inheritance and extensibility of objects. Function objects store data that describe information about their corresponding functions, such as their parameters, scope, and code. Figure 4.1 shows the table in the standard that describes the internal slots of function objects. It is worth mentioning the key slots of those Function Objects:

- The *Environment* slot stores the outer scope in which the function was declared.
- The *FormalParameters* slot stores the list containing the names of the formal parameters of the function.
- The *ECMAScriptCode* slot stores the AST node representating the function body.
- The *ThisMode* slot stores a string that defines how the `this` keyword will be interpreted within the function body.

Table 27 — Internal Slots of ECMAScript Function Objects

Internal Slot	Type	Description
[[Environment]]	Lexical Environment	The Lexical Environment that the function was closed over. Used as the outer environment when evaluating the code of the function.
[[FormalParameters]]	Parse Node	The root parse node of the source text that defines the function's formal parameter list.
[[FunctionKind]]	String	Either "**normal**", "**classConstructor**" or "**generator**".
[[ECMAScriptCode]]	Parse Node	The root parse node of the source text that defines the function's body.
[[ConstructorKind]]	String	Either "**base**" or "**derived**".
[[Realm]]	Realm Record	The Code Realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.
[[ThisMode]]	(lexical, strict, global)	Defines how **this** references are interpreted within the formal parameters and code body of the function. **lexical** means that **this** refers to the **this** value of a lexically enclosing function. **strict** means that the **this** value is used exactly as provided by an invocation of the function. **global** means that a **this** value of **undefined** is interpreted as a reference to the global object.
[[Strict]]	Boolean	**true** if this is a strict mode function, **false** if this is not a strict mode function.
[[HomeObject]]	Object	If the function uses **super**, this is the object whose [[GetPrototypeOf]] provides the object where **super** property lookups begin.

Figure 4.1: The attributes of Function Objects in ES6

In order to create a function object, the standard has a set of internal functions that handle different aspects of function creation. Figure 4.2 shows a sequence diagram describing the internal methods that participate in the interpretation of a function declaration. The process begins with the `FunctionDeclaration` internal function, which is responsible for interpreting function declarations. This function calls the internal function `FunctionCreate`, which is in charge of creating function objects. `FunctionCreate` calls the auxiliary function `FunctionAllocate` and `FunctionInitialize`. The former sets more general internal slots such as `Strict` and `FunctionKind`, while the latter sets the internal slots related to the function itself, such as `FormalParameters`, `Environment`, and `ECMAScriptCode`.

After the return from `FunctionCreate`, two additional calls may be made, though they are optional. The `MakeConstructor` method converts a function object into a Constructor, and `SetFunctionName` associates the name of the function with the newly created function object (if the function is not anonymous).

To complement the function creation process, let us illustrate our implementation of ECMARef6 and demonstrate its compliance with the standard text. Figure 4.3 displays our implementation of the `FunctionInitialize` function. Each line, implemented using the ECMA-SL language, has a comment above with the corresponding line of the standard. The function has five parameters: the function object F, the kind of function, the parameter list, the body of the function, and the scope. In the following, we give a brief description of the behaviour of this function:

- Instructions 1-3 set the property `length` of the function object to the number of provided parameters;
- Instruction 4 simply makes sure that the assignment to `length` was successful;
- Instructions 5-8 set the internal fields of the function object using the provided parameters `Environment`, `FormalParameters`, `ECMAScriptCode`;

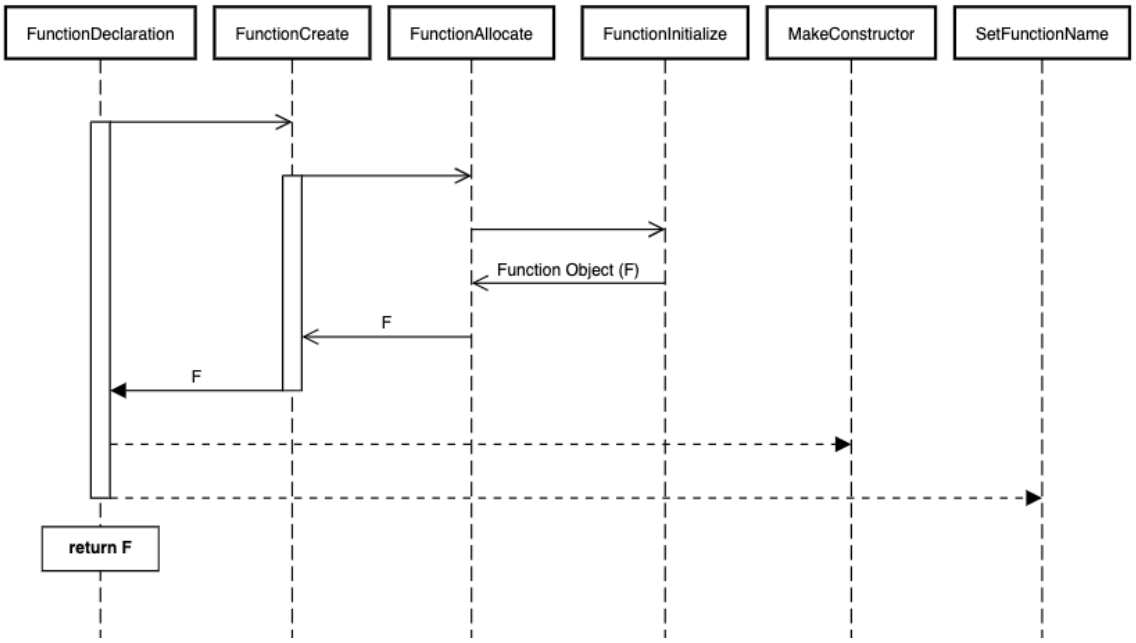


Figure 4.2: A sequence diagram that creates a Function Object

- Instruction 9-11 set the ThisMode of the function object based on the function kind.
- Instruction 12 simply returns the now completed Function Object.

```
function FunctionInitialize(F, kind, ParameterList, Body, Scope) {
  /* 1. Assert: F is an extensible object that does not have a length own
     property. */
  assert(F.Extensible = true && !("length" in_obj F));
  /* 2. Let len be the ExpectedArgumentCount of ParameterList. */
  len := ExpectedArgumentCount(paramsDetails)
  /* 3. Let status be DefinePropertyOrThrow(F, "length",
     PropertyDescriptor{[[Value]]: len, [[Writable]]: false,
     [[Enumerable]]: false, [[Configurable]]: true}). */
  status := DefinePropertyOrThrow(F, "length", newDataPropertyDescriptorFull(int_to_float len, false, false, true));
  /* 4. Assert: status is not an abrupt completion. */
  assert(!isAnAbruptCompletion(status));
  /* 5. Let Strict be the value of the [[Strict]] internal slot of F. */
  Strict := F.Strict;
  /* 6. Set the [[Environment]] internal slot of F to the value of Scope. */
  F.Environment := Scope;
  /* 7. Set the [[FormalParameters]] internal slot of F to ParameterList . */
  F.FormalParameters := ParameterList;
  /* 8. Set the [[ECMAScriptCode]] internal slot of F to Body. */
  F.ECMAScriptCode := Body;
  /* 9. If kind is Arrow, */
  if (kind = "Arrow") {
    /* set the [[ThisMode]] internal slot of F to lexical */
    F.ThisMode := "lexical"
  }
  /* 10. Else if Strict is true, */
  elif (Strict = true) {
    /* set the [[ThisMode]] internal slot of F to strict */
    F.ThisMode := "strict"
  }
  /* 11. Else set the [[ThisMode]] internal slot of F to global. */
  else {
      F.ThisMode := "global"
  };
  /* 12. Return F. */
  return F
};
```

Figure 4.3: The implementation for `FunctionInitialize` in ECMARef6

Figure 4.4 provides a visual representation of the Function Objects created by interpretation of functions `sum_one` and `sum_three` from Listing 4. Despite one declaration being a regular function and the other an arrow function, the objects are nearly identical. The only noteworthy difference is in the value of the internal slot `ThisMode`. Functions declared using the function keyword have either the `global` or `strict` flags for the `ThisMode` internal slot, while arrow functions have the `lexical` flag. These flags are used to determine the binding of the `this` keyword inside the body of the declared function.

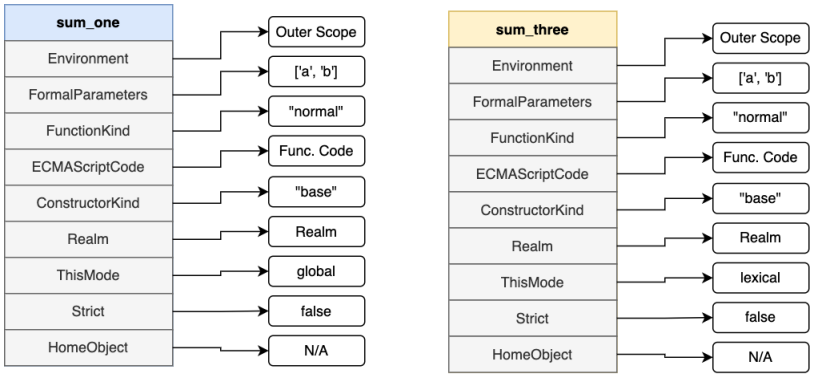


Figure 4.4: A visual representation of Function Objects

4.1.2 Function Execution

The second stage of the function lifecycle is the execution phase, during which the code stored in the `Function Object` is executed with the necessary context, including parameters and scope. In this subsection, we will expand on how the interpreter executes functions and also how the execution affects the current scope of an ECMAScript program.

To illustrate the upcoming concepts, let us consider a more complex JavaScript example. Listing 5 declares a function called `makeIdGen`, that is used to create id generators. More concretely, `makeIdGen` takes a prefix parameter and returns another function that generates unique IDs by appending a counter to the given prefix. The returned function is an example of a closure because it has access to the `count` variable in its parent function (`makeIdGen`). Specifically, each time the returned function is called, it increments `count` and creates a new ID by combining the given prefix with the current value of `count`. Finally, the new ID is returned to the caller.

Using the function `makeIdGen`, the example then creates two ID generators, `gen_1` and `gen_2`. `gen_1` generates IDs with the prefix *prefix_one*, while `gen_2` generates IDs with the prefix *prefix_two*. When `gen_1()` is called the first time, it generates the ID *prefix_one_0*. Each subsequent call to `gen_1()` generates a new ID with the same prefix but an incremented count. For example, the second call to `gen_1()` generates the ID *prefix_one_1*. Similarly, when `gen_2()` is called the first time, it generates the ID *prefix_two_0*. Each subsequent call to `gen_2()` generates a new ID with the same prefix but an incremented count.

4.1.3 Scope Representation

To understand how the interpreter handles scope, it is important to understand the three main objects used by the interpreter to represent it: Execution Contexts, Lexical Environments, and Environment Records. In the context of function execution, these objects work together to determine which variables are accessible within a given function call. For example, consider the call to `gen_1` on line 13 of Listing 5. To execute this call, the interpreter follows these steps:

```
function makeIdGen(prefix) {
  let count = 0
  return () => {
    var id = prefix + '_' + count
    count++
    return id
  }
}

let gen_1 = makeIdGen('prefix_one')
let gen_2 = makeIdGen('prefix_two')
gen_1() // prefix_one_0
gen_1() // prefix_one_1
gen_2() // prefix_two_0
```

Listing 5: A more complex function example in ES6

- It creates a new Execution Context associated with the function execution (Execution Context 1).
- It creates a new Lexical Environment (Lexical Environment 1) to connect the current scope to the scope in which the function was declared (Lexical Environment 2) and stores it inside Execution Context 1.
- It creates a new Environment Record for storing the bindings of the parameters and local variables of `gen_1` (Environment Record 1) and stores it inside Lexical Environment 1.

Figure 4.5 provides a visual representation of the scope objects involved in the call. Importantly, the Lexical Environment of `gen_1` is connected to the Lexical Environment that was active when it was created, which corresponds to a Lexical Environment associated with the first call to `makeIdGen`. Therefore, when `gen_1` tries to access the value of variable `count` or `prefix`, the interpreter can simply go up one level in the chain of Lexical Environments to find the appropriate value.

One aspect that requires explanation is how the interpreter can access the lexical environment that was active when `gen_1` was created, at the time of calling the function. In order to accomplish this, the interpreter stores a reference to the lexical environment that was active during function creation within the function object itself. Specifically, the function object of `gen_1` contains a reference to the lexical environment of the `MakeIdGen` function that was active at the time of its creation.

Finally, it is important to note that at the top of every lexical environment chain, there is a lexical environment that is associated with the global object, rather than an environment record. This global object is where the bindings for global variables are stored.

4.1.4 Function Call Interpretation

Having understood how scope is represented in the interpreter, we are in a position to explain how function calls are executed. Since this process involves a large number of auxiliary functions, we only focus on the most relevant of those. Figure 4.6 represents the relevant fragment of the interpreter call graph that captures their evaluation order. The entry point of the call graph is the function JS_Interpreter_Expr used to interpreter JavaScript expressions. When given a CallExpression AST node, the function JS_Interpreter_Expr ¡performs a bunch of checks¿ and calls the auxiliary function `Call`. This function will, in turn, make use of the the functions `PrepareForOrdinaryCall`, for creating the new execution context, and `OrdinaryCallEvaluateBody`, for executing the body of the function in the created scope. Importantly, the `OrdinaryCallEvaluateBody` makes use of the auxiliary function

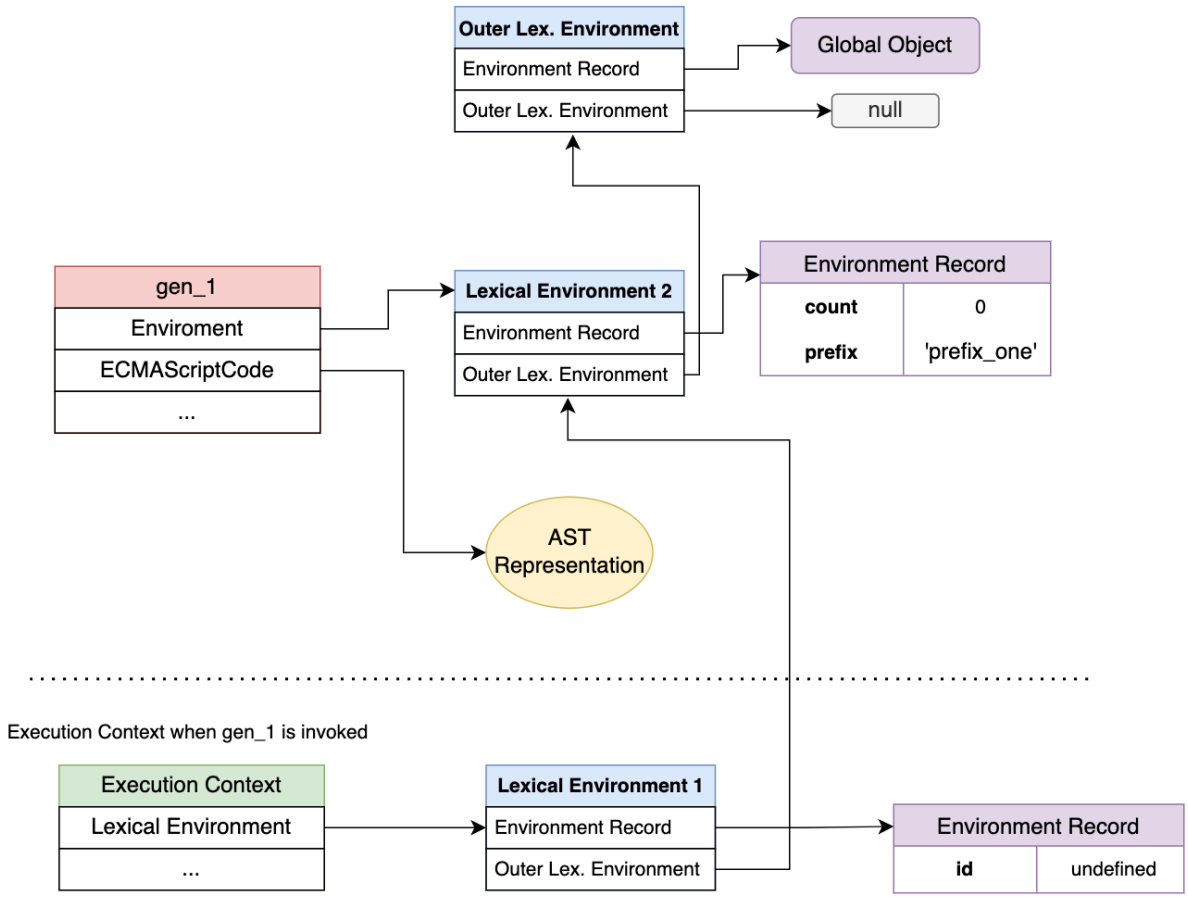


Figure 4.5: Scope Representation when `gen_1` is invoked

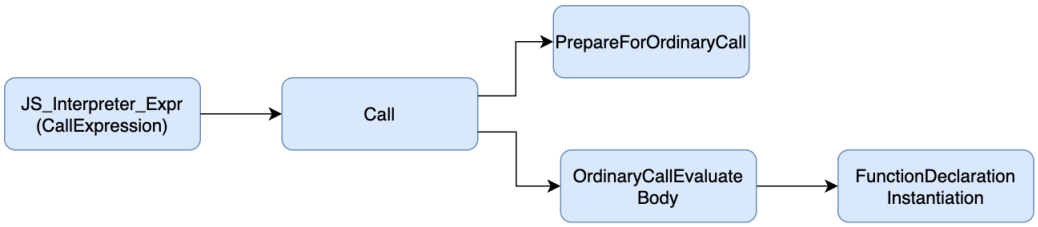


Figure 4.6: A simplified internal call graph for Function execution in ECMARef6

`FunctionDeclarationInstantiation` for setting up the lexical environment and environment record of the new function with all the necessary bindings. In the following, we will detail our implementation of these four function, as they are an essential part of our reference implementation.

Call The Call function illustrated in Figure 4.7 takes two parameters: the *thisArgument*, which is the this value used for the function, and the *argumentsList*, which is the list of parameters passed to the function execution. Line 1 checks if the object associated with that function (F) is a Function Object. Line 2 checks if the function is not a class constructor. Lines 3 to 6 prepare the context for the function execution using the the auxiliary functions `PrepareForOrdinaryCall` and `OrdinaryCallBindThis`. Line 7 evaluates the function body through the use of the `OrdinaryCallEvaluateBody` function. Line 8 restores the scope to be the one before the function execution. Lines 9 to 11 determine what will be returned from the function: either the return value if the function body has a return statement or *undefined*, which is the default return value for functions without the return statement.

[[Call]] (thisArgument, argumentsList)

The [[Call]] internal method for an ECMAScript function object *F* is called with parameters *thisArgument* and *argumentsList*, a List of ECMAScript language values. The following steps are taken:

1. Assert: *F* is an ECMAScript function object.
2. If *F*'s [[FunctionKind]] internal slot is **"classConstructor"**, throw a **TypeError** exception.
3. Let *callerContext* be the running execution context.
4. Let *calleeContext* be PrepareForOrdinaryCall(*F*, **undefined**).
5. Assert: *calleeContext* is now the running execution context.
6. Perform OrdinaryCallBindThis(*F*, *calleeContext*, *thisArgument*).
7. Let *result* be OrdinaryCallEvaluateBody(*F*, *argumentsList*).
8. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
9. If *result*.[[type]] is return, return NormalCompletion(*result*.[[value]]).
10. ReturnIfAbrupt(*result*).
11. Return NormalCompletion(**undefined**).

Figure 4.7: The specification of the `Call` internal function from the ES6

PrepareForOrdinaryCall The main task of the PrepareForOrdinaryCall internal function is to create the new context in which the function will be executed. In order to achieve that, first the interpreter creates a new Execution Context and associates with a new Function Environment. A Function Environment is a special kind of Lexical Environment associated with a specific Function Object. This new Function Environment will be stored in the Environment internal slot of the Function Object, i.e, the scope in which the function was created. After all objects are created, the interpreter then suspends the current Execution Context and makes sure the running execution context is the newly created one.

OrdinaryCallEvaluateBody The OrdinaryCallEvaluateBody internal function has only two steps: executing the FunctionDeclarationInstantiation internal function and evaluating the function body, which is stored in the ECMAScriptCode internal slot of the Function Object. The FunctionDeclarationInstantiation internal function is arguably one of the most complex internal functions in the entire ECMAScript standard and deserves separate explanation.

Once the execution context is properly set up by FunctionDeclarationInstantiation, the final step is to execute the function body. This step is relatively simple, mainly because we store the Abstract Syntax Tree (AST) of the body on the Function Object. Hence, the interpreter simply has to recursively call itself with the AST of function's body as an argument.

FunctionDeclarationInstatiation This internal function is responsible for taking the newly created context and inserting all the necessary bindings into the Environment Record to ensure the proper execution of the function. *FunctionDeclarationInstantiation* in the Standard is specified in over 40 lines of pseudocode, as it involves adding more than just the arguments passed to the function execution to the context. This includes elements such as:

- Variables: every variable defined in the body of the function is initialized with the value *undefined*;
- Parameters: Every parameter is initialized with either the value it was called with or with *undefined* in case no value was passed.
- Arguments Object: In some cases, ECMAScript initializes a special variable called `arguments`, which is an array containing all the parameters with which the function was called.

- Parameter expressions: In ES6, parameters can contain expressions. These expressions are evaluated and assigned to the associated parameters. We will provide a more detailed explanation of this type of operation in Section 4.2.

- Functions: Functions declared inside the function body are initialized before the execution, meaning that the interpreter creates the Function Objects associated with those functions even before the execution starts.

FunctionDeclarationInstatiation is complex due to the number of tasks it performs. Each of these tasks must be completed correctly, and the interpreter must be cautious with errors and possible edge cases, such as repeated parameter names. Another behaviour of the standard that this function must implement is the hoisting of variables and inner functions. Hoisting is an ECMAScript mechanism where variables and functions can be used before they are declared, regardless of where they are actually written in the function body.

4.2 Assignment Patterns

The transition from ECMAScript 5 to ECMAScript 6 also introduced *assignment patterns* - a powerful feature for directly assigning internal elements of ECMAScript values such as arrays and objects. These patterns can be used both for variable assignments and as function parameters. Often referred to as destructuring expressions, they allow extracting specific parts of arrays and objects, making code more concise and readable. In this section, we will dive deeper into these new patterns and explore how they are implemented in ECMARef6.

An assignment pattern comprises two elements: the left side, which describes the pattern to be applied, and the right side, which is the value that the pattern should be applied to. A straightforward example is `var [first,..] = [1,2]`. In this case, the first element of the array, which is 1, is assigned to the variable `first`, which is declared inside an *Array Pattern* pattern on the left. Let us examine all patterns that can be used on the left side:

- **Array Pattern**: As demonstrated in the example above (`var [first] = [1,2]`), Array Patterns enable developers to assign specific values from an array to a variable based on their position.

- **Object Pattern**: Object Patterns enable the developer to associate a specific object key with a variable named after that key. For instance, the pattern `var a = a: 1` assigns the value of the property key `a` (e.g., 1) to variable `a`.

- **Rest Element**: This pattern can be used in arrays and objects to assign all remaining values of the corresponding element to a given variable. For instance, the variable declaration `var [first, ...rest] = [1,2,3,4]` assigns variable `first` to 1 variable `rest` to an array containing all the remaining values of the right-hand-side array (e.g., `[2,3,4]`).

- **Assignment Pattern**: The standard uses the term assignment patterns for assignment expressions where the left-hand-side expression has nested patterns. For instance, the variable declaration `var {a: [first]} = {a: [2]}` contains an array pattern within an object pattern. This declaration assigns `first` to 2 and leaves the `a` undefined. We will expand on this case with more examples later in this section.

- **Identifier**: This is the simple assignment case in which we have a name on the left and a value on the right. For instance, `var a = 1` is an example of an assignment pattern with an identifier.

Assignment Patterns in Function Parameters As mentioned earlier, assignment patterns are not only used in variable assignments but also in function parameters. In this case, the left side is the pattern with which the function was defined, and the right side is the value passed as the parameter when the function is called. Since Assignment Patterns can be nested, functions can now have default parameter values. For instance, consider the simple example in Listing 6, where the function `get_age` declared in line 1 has an Assignment Pattern as the argument using the identifier case: `age = 20`. When we call the function in line 4, the interpreter understands that no value is assigned to `age`, so it relies on the value 20 that is in the pattern. In line 5, we call the function with the value 30; in this case, the interpreter would assign 30 to `age` and ignore the default value.

```
1  function get_age(age = 20) {
2          return age
3  }
4
5  get_age()   // Output: 20
6  get_age(30) //Output: 30
```

Listing 6: A simple example of default parameters in ES6 functions

```
1  function person_details(
2  {name, age = 20} = {name: 'John', age: 20}, [first, second], ...otherParameters)
3    {
4      console.log('Name', name)
5      console.log('Age', age)
6      console.log('First', first)
7      console.log('Second', second)
8      console.log('Other Parameters', otherParameters)
9    }
10
11 personDetails({name: 'Dave', age: 35}, ['Howdy'], 'extra1', 'extra2')
12 // Output: Name Dave
13 //         Age 35
14 //         First Howdy
15 //         Second undefined
16 //         Other Parameters ['extra1', 'extra2']
17
18 personDetails({name: 'Rafael'}, [1, 2], 3)
19 // Output: Name Dave
20 //         Age 20
21 //         First 1
22 //         Second 2
23 //         Other Parameters [3]
24
25 personDetails({ })
26 // Output: Name undefined
27 //         Age 20
28 //         First undefined
29 //         Second undefined
30 //         Other Parameters []
```

Listing 7: A more complex example of Assignment Patterns usage in functions

Consider Listing 7, which explores a more complex example of the usage of assignment patterns in function parameters. In particular, it defines a function `person_details` with four parameters, each cor-

responding to an assignment pattern. Below, we discuss each of the parameters individually, explaining how they are interpreted in the three function calls that follow the code of the function.

1. *Assignment Pattern* The first parameter is an Assignment Pattern whose left side is an Object Pattern, intended to extract the bindings of variables `name` and `age` from the supplied argument. The `age` binding has a default value of 20. The outer Assignment Pattern has a default value of {`name:'John', age:20`}. This example illustrates that default values can occur at multiple nesting levels of assignment patterns. Let us now take a look at how this pattern is evaluated on the subsequent calls to function `person_details`.

   In the first call (line 11), the assignment pattern must be matched against the object {`name: 'Dave', age: 35`}, which contains the two keys of the left-hand-side object pattern. Hence, the semantics destructs the supplied object, assigning the the values *'Dave'* and *35* to the variables `name` and `age`, respectively.

   In the second call (line 18), the assignment pattern must be matched against the object {`name: 'Rafael'`}, which contains only one key of the left-hand-side object pattern. Hence, the semantics destructs the supplied object, assigning the value *'Rafael'* to the variable `name`. The variable `age` is set to the default value, 20.

   In the last call (line 25), the assignment pattern must be matched against the empty object {}. In this case, the variable `age` is set to its default value, 20, whereas the variable `name` is set to `undefined`.

2. *Array Pattern* The second parameter is an Array Pattern intended to extract bindings of variables `first` and `second` from the supplied array argument.

   In the first call (line 11), the array pattern must be matched against the array value `['Howdy']`, which only contains one element of the left-hand-side array pattern. Hence, the semantics destructs the supplied array, assigning the value *'Howdy'* to the variable `first` and setting `second` to *undefined*.

   In the second call (line 18), the array pattern must be matched against the array value `[1,2]`, which contains both elements of the array pattern. Hence, the semantics destructs the supplied array, assigning the values *1* and *2* to `first` and `second`, respectively.

   In the last call (line 25), the array pattern must be matched against *undefined*, as no value was supplied for the second parameter. In this case, both `first` and `second` are set to *undefined*.

3. *Rest Pattern* The third and last parameter is a rest pattern associated with variable `otherParameters`. When used in functions, the rest element aggregates all remaining parameters that were supplied to the function in an array.

   In the first call (line 11), the rest element is matched against the two remaining parameters: `'extra1'` and `'extra2'`. In this case, the `otherParameters` variable is set to the array value ['extra1', 'extra2'] that contains all remaining parameters with which the function was called.

   In the second call (line 18), the rest element is matched against the only remaining parameter: `3`. In this case, the `otherParameters` variable is set to the array value `[3]`.

   In the last call (line 25), the rest element is matched against `undefined`, as all parameters were already interpreted. In this case, the `otherParameters` variable is assigned to an empty array `[]`.

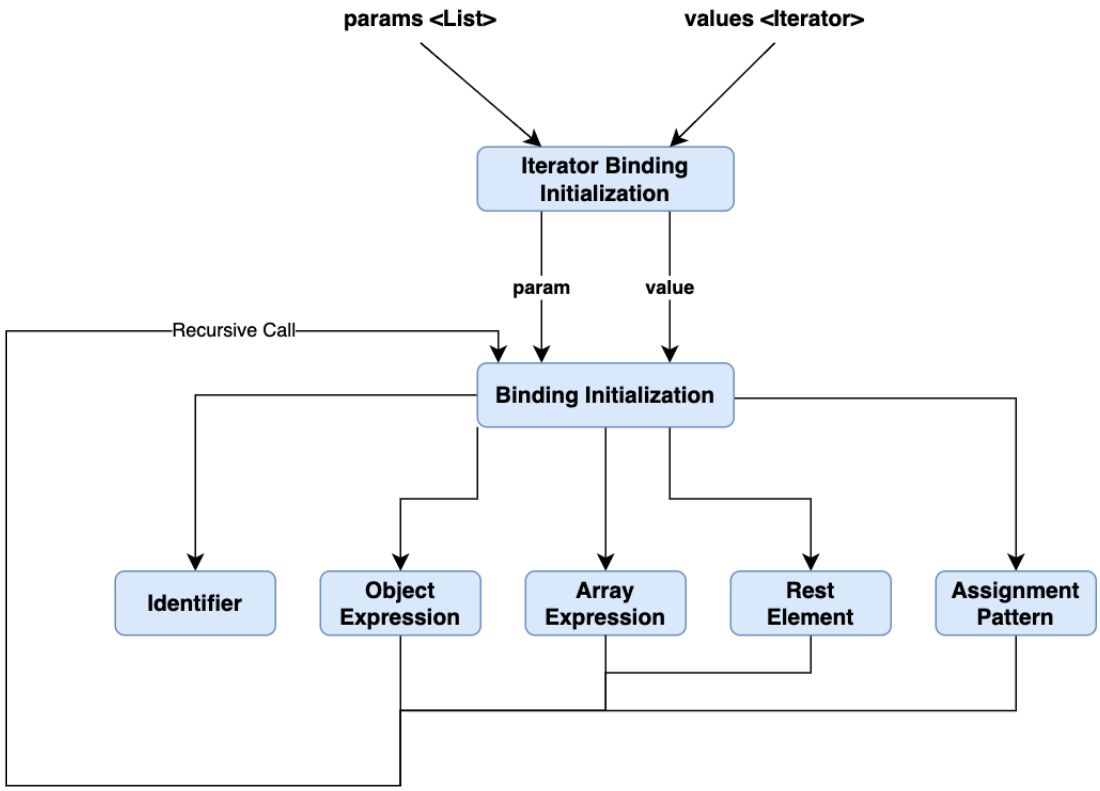


Figure 4.8: A Visual representation of our assignment pattern algorithm

4.2.1 Assignment Patterns Implementation

The specification of this feature in the standard can be challenging to comprehend because of the potential for recursion. Although the standard endeavors to interpret a single Assignment Pattern in small steps, the interpretation is obscured by several internal function calls. This complexity resulted in confusion, and we opted to deviate from the standard in our implementation of the feature. Our objective was to present a more straightforward approach that would make the functionality clear and understandable.

To illustrate our implementation, let us examine Figure 4.8, a simplified diagram that represents the internal function used to interpret assignment patterns of function parameters. The top-level function, `IteratorBindingInitialization`, receives a list of pattern parameters and an iterator that contains the values supplied to the the function. By utilizing the iterator, we can guarantee that each parameter in the list refers to the correct value. This is achieved by executing a step on the iterator only when the previous parameter was fully interpreted.

The `IteratorBindingInitialization` function iterates through the parameters and, for each parameter and its associated value, calls the recursive function `BindingInitialization`. This function examines the pattern's type and invokes the relevant function responsible for its interpretation. The identifier pattern serves as the base case for `BindingInitialization`, with all other functions calling `BindingInitialization` again and modifying the value if required, specifically if there's a nested assignment pattern.

Now that we have a high-level view of the implementation, let us examine it in greater detail. Listings 8 and 9 show the pseudocode of the functions referred to in Figure 4.8. It is worth noting that we made the pseudocode simpler than the actual implementation for presentation purposes. The real implementation

```
function IteratorBindingInitialization(params, valuesIterator, scope) {
  for (param in params) {
    value := getIteratorValue(valuesIterator)
    BindingInitialization(param, value, valuesIterator, scope)
  }
}

function BindingInitialization(param, value, valuesIterator, scope) {
  match param with {
    { type: 'Identifier' } : {
     BindingInitializationIdentifier(param, value, scope)
    },
    { type: 'ObjectExpression' } : {
      BindingInitializationObject(param, value, scope)
    },
    { type: 'ArrayExpression' } : {
      BindingInitializationArray(param, value, scope)
    },
    { type: 'RestElement' } : {
      BindingInitializationRest(param, valuesIterator, scope)
    },
    { type: 'AssignmentPattern' } : {
      BindingInitializationAssignment(param, value, scope)
    }
  }
}
```

Listing 8: Pseudo-code for the high-level functions involved in Assignment Patterns

is substantially more complex as it must be compliant with the standard and handle all possible edge cases and errors. Nonetheless, the pseudocode is useful to get a general understanding of the overall structure and control flow of our implementation.

Listing 8 presents the pseudocode for the highest-level functions: `IteratorBindingInitialization` and `BindingInitialization`. The former iterates through all parameters, invoking `BindingInitialization` on each parameter pattern with the corresponding value and scope. `BindingInitialization` uses a match statement to route the implementation to the appropriate function handler, depending on the type of parameter pattern.

Listing 9 presents the pseudocode for the specific implementations of those handler functions. The first one in lines 1 to 3 is `BindingInitializationIdentifier`, which is the base case. In this case, we have the binding name and value, and we just need to initialize that binding with its associated value in the current scope. With the base case covered, we can now analyze the other more complex cases in detail:

- **BindingInitializationObject**: This function is used to match object patterns against their associated object values and its pseudo-code is given in lines 5-10. The function receives an object pattern, `objPat`, an object value, `objValue`, and the current scope, `scope`. The logic is as follows: for each of the keys in the object pattern, the function gets the associated key in `objValue` and recursively calls the `BindingInitialization` function to match the corresponding key pattern to its associated value. When a key is present in `objPat` but not in `objValue`, it is initialized as *undefined*. Only the keys present in `objPat` are initialized, so any keys in `objValue` that are not in `objPat` are ignored.

- **BindingInitializationArray**: This function is used to match array patterns against their associated

```
1  function BindingInitializationIdentifier(param, value, scope) {
2    scope.InitializeBinding(param, value)
3  }
4
5  function BindingInitializationObject(objPat, objValue, scope) {
6    for (key in param) {
7      value := objValue[key]
8      BindingInitialization(key, value, scope)
9    }
10 }
11
12 function BindingInitializationArray(arrPat, arrValue, scope) {
13   for (index i in arrPat) {
14     value := getArrayValue(arrValue, i)
15     BindingInitialization(arrPat[i], value, scope)
16   }
17 }
18
19 function BindingInitializationRest(restIdentifier, valuesIterator, scope) {
20   allValues := getAllValues(valuesIterator)
21   BindingInitialization(restIdentifier, allValues, scope)
22 }
23
24 function BindingInitializationAssignment(nestedPattern, currentValue, scope) {
25         if (currentValue == undefined) {
26           value := nestedPattern.right
27     } else {
28           value := currentValue
29         }
30         BindingInitialization(nestedPattern.left, value, scope)
31 }
```

Listing 9: Pseudo-code for the specific handler involved in the Assignment Patterns

array values and its pseudo-code is given in lines 12-17. The function receives an array pattern, `arrPat`, an array value, `arrValue`, and the current scope, `scope`. The logic is as follows: for each index `i` in `arrPat`, the corresponding value for that index in `arrValue` is assigned to the variable `value`. Then, the function recursively calls the `BindingInitialization` function to match the pattern stored in that index in `arrPat` with its associated `value`. If `arrValue` does not contain an element at that index, the interpreter assumes the value to be *undefined*. The values within the array `arrValue` that do not exist in the array `arrPat` are ignored.

- **BindingInitializationRest**: This function is used to match rest elements against their associated values and its pseudo-code is given in lines 19-22. The function receives a binding used to store the final value, `restIdentifier`, an iterator with all remaining values, `valuesIterator`, and the current scope, `scope`. The logic is as follows: all values still remaining on the iterator are obtained using the function `getAllIteratorValues` and its result is assigned to the variable `allValues`. After that, the function recursively calls `BindingInitialization` to match the given binding with the obtained value in `allValues`. If the iterator is already done and no values remain, the interpreter simply sets `allValues` to an empty array.

- **BindingInitializationAssignment**: This function is used to match nested assignment patterns against their associated values and its pseudo-code is given in lines 24-31. The function receives the nested pattern, `nestedPattern`, the current value being matched to the pattern, `currentValue`,

and the current scope, `scope`. The logic is as follows: if the current value is *undefined*, then the new current value is the right-hand side of the nested pattern; otherwise, the current value remains unchanged. The function then recursively calls `BindingInitialization` to match the left-hand side of the nested pattern to the current value. This approach guarantees the behavior of default values.

Despite the examples focusing on function parameters, this architecture can easily be extended to variable assignments. In this case, instead of calling `IteratorBindingInitialization`, the interpreter would call `BindingInitialization`, which requires a pattern and its associated value. This demonstrates that our implementation of assignment patterns, while deviating from the standard, still satisfies all use cases and is sufficient for the implementation of this feature.

4.3 Classes

ECMAScript 6 introduced the concept of classes as a new feature to the language. Prior to that, classes could only be emulated through object prototypes. Classes are a fundamental concept in object-oriented programming that allow developers to create blueprints for objects with shared characteristics and behaviors. Classes also allow developers to use inheritance in a more standard way than the prototype base inheritance present in the language. In this chapter we will expand on the main functionalities of classes and their implementation in ECMARef6.

In ECMAScript, an object instantiated from a class becomes an instance of that class and inherits all properties and methods defined within it. Although each object has its distinct set of properties and behaviors, they share the same structure and functionality as other instances of the same class. To gain a better understanding of classes in ECMAScript, let us explore the key aspects defined by the standard:

- **Constructor**: Class constructors are identified by the name 'constructor' and are executed when new objects of their respective classes are created. Constructors are used to initialize the object's properties and can also accept arguments to set initial values.
- **Inheritance**: Inheritance is the ability of a class to inherit properties and methods from a parent class or any object. This means that objects instantiated from the child class will have access to properties and methods defined not only within the child class but also within its parent class and the entire inheritance chain.
- **Super**: In ES6 classes, the super keyword is used to call the constructor or a method of the parent class from the child class. When a class extends another class, the subclass inherits all properties and methods of the parent class. The `super` expression is used to access these inherited properties and methods and can be called within the constructor or any method of the child class.
- **Static Methods**: A static method is a method that belongs to the class itself, rather than to any instance of the class. This means that a static method can be called directly on the class without the need to create an instance of the class. Static methods are typically used for utility functions or operations that do not require access to instance-specific data.
- **Instance methods**: In ES6 classes, an instance method is a method that belongs to each instance of the class, rather than the class itself. This means that an instance method can only be called on an instance of the class and has access to instance-specific data. These methods are typically used for operations that are specific to each instance of the class, such as manipulating or retrieving data from the instance's properties.

- **This**: The `this` keyword is used within classes to refer to the current instance of the class, allowing methods to access and modify instance-specific data.

To illustrate the usage of these features, let us examine the example in Listing 10. In line 1, we define a class using the `class` keyword and name it `Shape`. Lines 2 to 5 define the `constructor` function, which takes two arguments, `x` and `y`. The function then assigns the parameter values to instance properties `x` and `y` using the `this` keyword. After that, we define a dynamic method called `getCoordinates`, which logs `x` and `y` in a formatted way. The last method defined in the `Shape` class is the static method `shape_info`. This method simply logs a pre-established message.

In the second part of Listing 10, we have the definition of the `Circle` class. This definition happens on lines 16-30, and in this case, the `Circle` class extends the `Shape` class, meaning that `Shape` is the parent of `Circle`. On lines 17-20, we define the constructor function for `Circle`. In this case, the constructor takes three arguments: `x`, `y`, and `radius`. Line 18 in the constructor calls `super` with `x` and `y`, meaning that it will execute the constructor of the parent class with arguments `x` and `y`. After that, on line 19, we set `this.radius` to the value of the `radius` parameter. The `Circle` class defines two methods. The first one is `getArea`, which returns the area of the circle on which it is called. The second method is static and is called `info`. The definition of the method on lines 26-28 shows us that this method will call the `info` method from the parent using `super.info()` and then log an additional message.

Now that we have all the definitions, let us take a look at the last part of the example, which is the usage of both classes. On line 32, we instantiate a circle using the `new` keyword. This instance is created with the arguments `(0, 0, 5)` and is assigned to the variable `circle`. On line 33, we call the `getArea` method of the newly created circle, obtaining the result `78.53`, which is shown as a code comment. The next operation calls the `getCoordinates` method of the parent class. In this case, the answer is `(0, 0)`, which is also shown as a code comment. On line 35, we call the `info` method on the `circle` instance, obtaining an error that says that `info` is not callable. This happens because the instance does not have the `info` method; it is only present in the class object itself. The last line calls `info` in the class object, and for this case, we would obtain logs for the messages associated with both `Circle` and `Shape`.

4.3.1 Class Hierarchy Representation

Classes in ES6 are not a complete replacement for prototype-based inheritance; instead, they provide a convenient syntactic layer that builds upon the existing mechanisms of the language for representing functions and implementing prototype-based inheritance. When a class is defined, it is ultimately evaluated in terms of its constructor and its associated objects.

Each class is associated with two distinct objects: the class constructor, which is a function object, and the object linked to the constructor's `prototype` field. It is important to note that this object corresponds to the internal prototype of instances of that class, and for simplicity, we will refer to this object as the *prototype object*. These two objects store the different methods associated with the class: instance methods are stored in the prototype object, while static methods are stored directly in the class constructor. Figure 4.9 depicts the internal representation of the Shape and Circle classes, with the constructors illustrated in yellow and the prototype objects in green. From the diagram, we can observe the following:

- the instance methods of the Shape class, such as `getCoordinates`, are stored in the *Shape Prototype* object;
- the instance methods of the Circle class, such as `getArea`, are stored in the *Circle Prototype* object;
- the static methods of the Shape class are stored in the *Shape Constructor* object; and

```
class Shape {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  getCoordinates(){
    console.log('(', this.x, ', ', this.y, ')')
  }

  static shape_info() {
    console.log('Shapes are geometrical figures with no specific dimensions.');
  }
}

class Circle extends Shape {
  constructor(x, y, radius) {
    super(x, y);
    this.radius = radius;
  }

  getArea() {
    return Math.PI * this.radius ** 2;
  }

  static info() {
    super.shape_info();
    console.log('Circles are a type of shape with a constant curvature.');
  }
}

const circle = new Circle(0, 0, 5);
circle.getArea(); // Output: 78.53981633974483
circle.getCoordinates() // Output: (0, 0)
circle.info() // Error: info is not callable

Circle.info();
/* Output:
Shapes are geometrical figures with no specific dimensions.
Circles are a type of shape with a constant curvature.
*/
```

Listing 10: A example of Class usage and its features in ES6

- the static methods of the Circle class are stored in the *Circle Constructor* object.

This representation ensures that the prototype object directly serves as the prototype of instances belonging to the class. The diagram shows two instances represented in blue with their `[[Prototype]]` internal slots pointing to the corresponding prototype objects. Consequently, when a new circle is instantiated, for example, it has access to the `getArea` function defined in the *Circle Prototype* object. Furthermore, this internal representation ensures that the class objects contain all the static methods. This is possible because the class object itself serves as the constructor object, which holds all the static methods.

To understand how subclasses have access to the methods defined in their superclasses, we need to grasp how class-base inheritance is modeled in terms of prototype-based inheritance.

Representation of Class Inheritance There are two important aspects to consider when it comes to class-based inheritance: inheritance of static methods and inheritance of instance methods. These are

modeled differently.

Inheritance of instance methods relates to the need for instance methods of the superclass to be accessible from instances of the subclass. For example, an instance of *Circle* must be able to call the `getCoordinates` method of the *Shape* class. To ensure this functionality, we need the prototype object of the subclass to point to the prototype object of the superclass. In Figure 4.9, the *Circle Prototype* object in green has its internal slot `[[Prototype]]` pointing to *Shape Prototype*. Therefore, when searching for the `getCoordinates` method in an instance of a circle, the language semantics will first attempt to find it in the *Circle Prototype* object. If it is not there, it will continue following the prototype chain and attempt to locate it in the *Shape Prototype* object, where it will be found.

Inheritance of static methods refers to the mechanism that allows a subclass to invoke a static function defined in its superclass. For instance, the Circle class does not have the static method `shape_info`, but since this method exists in the Shape class, the expression `Circle.shape_info()` should evaluate to the Shape class's method. Once again, to ensure this functionality, we utilize the pre-existing mechanisms of prototypical inheritance in the language. In this case, we need to ensure that the *Circle Constructor* object has the *Shape Constructor* object as its prototype. In the diagram, we can observe that the *Circle Constructor* object has its `[[Prototype]]` internal slot pointing to *Shape Constructor*.

4.3.2 Classes Without Constructors

The implementation of classes in ECMAScript 6 required significant attention, not only to ensure proper internal representation but also due to some cases where the specification was unclear. A particular example is when a class does not define a constructor. In this subsection, we will delve into the specification of this case in ES6 and explore the implications of omitting a constructor in a class definition.

The interpretation of classes that have no associated constructor is particularly complex. As explained in Section 4.3.1, all static methods are tied to the constructor function representation. If a constructor is not explicitly defined in the class, the language should create one implicitly to ensure that static methods can be accessed by any class on that inheritance line. The ECMAScript standard therefore specifies a way of automatically assigning a constructor to a class that is defined without one.

Figure 4.10 displays the excerpt from the ECMAScript standard that specifies the behavior of empty constructors. There are two cases to consider depending on whether or not the current class has a parent class:

- **The current class has a parent:** The constructor should be the result of the evaluation of the following source text: `constructor(...args) super (...args);`. This constructor calls the constructor of the parent class, passing any arguments received by the child constructor.
- **The current class does not have a parent:** The constructor should be the result of the evaluation of the following source text: `constructor( ){ }`. This is an empty constructor, which does not take any arguments or perform any operations.

However, the standard does not provide a clear definition of how this interpretation should occur or how the text should be parsed at interpretation time. This leaves room for potential differences in implementation across different ECMAScript engines and the only reliable way to ensure compliance is through testing with Test262. As such, it is crucial to devote sufficient attention to this step to ensure that the program behaves as expected and conforms to the intended standard.

To address the issue of classes being defined without constructors in ECMAScript, we introduced a pre-processing step during the parsing of the program into the Abstract Syntax Tree (AST). Whenever a new class is created, we check if it has a constructor. If it does not, we add the necessary AST node to create a default constructor. Figure 4.11 illustrates our implementation of this pre-processing step. The

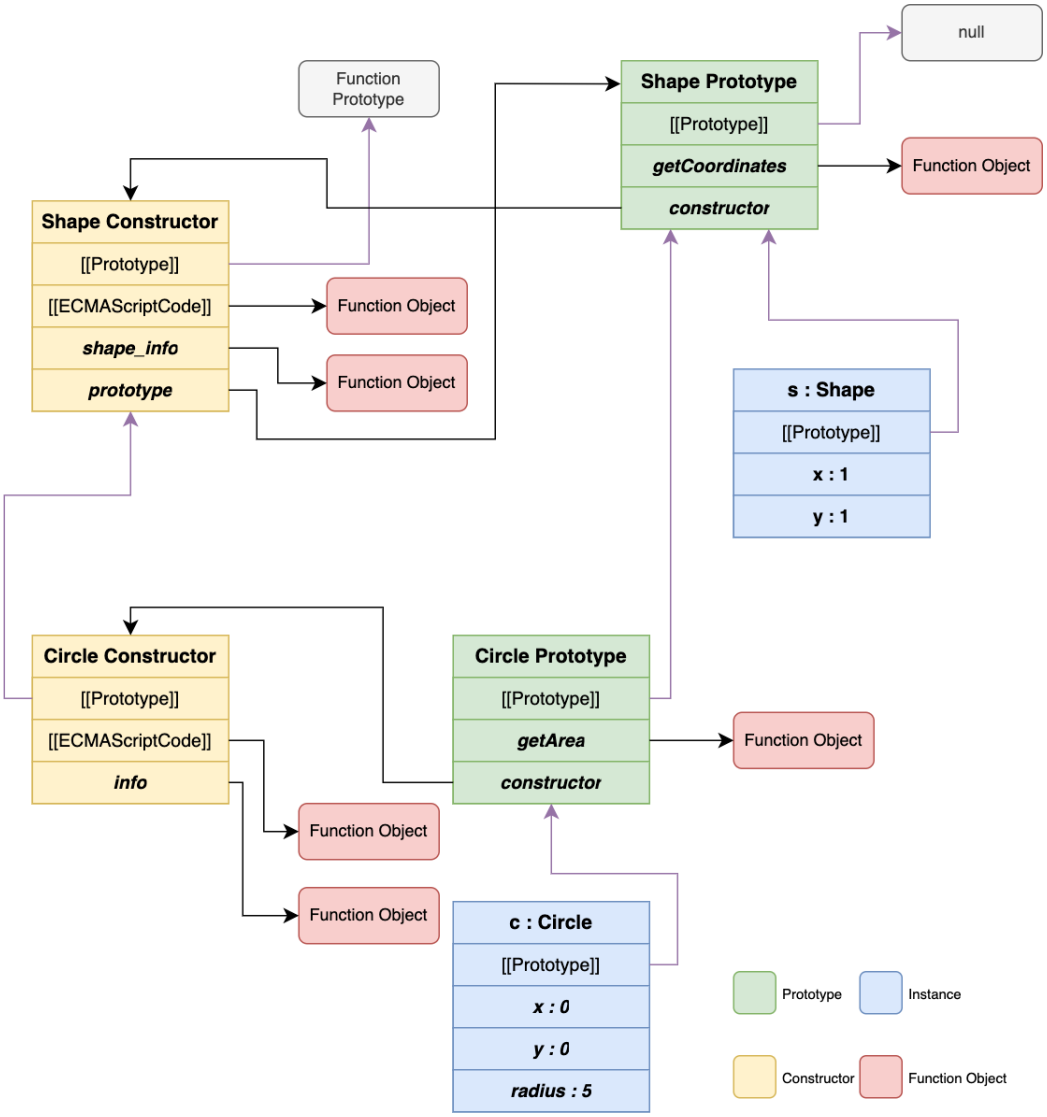


Figure 4.9: The internal representation of Class objects in ES6

10. If *constructor* is empty, then,
    a. If $ClassHeritage_{opt}$ is present, then
        i. Let *constructor* be the result of parsing the source text
            `constructor(... args){ super (...args);}`
        using the syntactic grammar with the goal symbol *MethodDefinition*.
    b. Else,
        i. Let *constructor* be the result of parsing the source text
            `constructor( ){ }`
        using the syntactic grammar with the goal symbol *MethodDefinition*.

Figure 4.10: The specification of functions without constructors

function defined on line 152 is executed for every class declaration and contains the logic for checking if a constructor is defined. If it is not, we use either the function `createConstructorWithSuper` or `createConstructor` to generate the appropriate AST node. The `createConstructor` function, displayed in line 159, returns the AST node that corresponds to the source text `constructor(){}`, thus ensuring

```
function addConstructor(obj) {
  if (noConstructor(obj) && hasSuper(obj)) {
    obj.body.body.push(createConstructorWithSuper());
  } else if (noConstructor(obj)) {
    obj.body.body.push(createConstructor());
  }
}
function createConstructor() {
  return {
    type: "MethodDefinition",
    key: {
      type: "Identifier",
      name: "constructor",
    },
    computed: false,
    value: {
      type: "FunctionExpression",
      id: null,
      params: [],
      body: {
        type: "BlockStatement",
        body: [],
      },
      generator: false,
      expression: false,
      async: false,
    },
    kind: "constructor",
    static: false,
  };
}
```

Figure 4.11: Our implementation to add the constructor function to classes

every declared class has a corresponding constructor function, and consistent behavior is guaranteed.

By using this approach, we were able to meet all the requirements. This example also makes it clear that implementing an interpreter goes far beyond simply transposing the lines of the standard. It requires a deep understanding of the language, and in addition, the sections where the standard is vague in terms of implementation require great attention and excellent integration with the language.

4.3.3 Super

The `super` keyword is a feature that was introduced in ECMAScript 2015 (ES6) to allow classes and subclasses in JavaScript to access and invoke functionality defined in their parent classes or objects. Its primary purpose is to facilitate inheritance and enable subclasses to access the constructor and methods of their parent classes or objects.

In ECMAScript, the `super` keyword has two primary uses: as a function call and as an object. When used as a function call, it allows us to invoke the constructor of the parent class within the child class. This ensures that the parent's constructor is executed before any additional code in the child constructor, enabling the child class to inherit properties and initialize the parent class's state.

On the other hand, when `super` is used as an object, it allows us to access properties or methods defined in the parent class from within the child class. This is particularly useful when the child class wants to extend or override specific behaviors of the parent class while retaining the rest of its functionality. By leveraging `super.propertyName` or `super.methodName()`, we can access and manipulate the parent's elements within the child class.

```
| { type: "SuperCall", arguments: Arguments } -> {
  /* 1. Let newTarget be GetNewTarget().*/
  newTarget := GetNewTarget(scope);
  /* 2. If newTarget is undefined, throw a ReferenceError exception. */
  if (newTarget = 'undefined) {
    throw ReferenceErrorConstructorInternal()
  };
  /* 3. Let func be GetSuperConstructor(). */
  func := GetSuperConstructor(scope);
  /* 4. ReturnIfAbrupt(func). */
  @ReturnIfAbrupt(func);
  /* 5. Let argList be ArgumentListEvaluation of Arguments. */
  argList := JS_Interpreter_Arguments(Arguments, scope);
  /* 6. ReturnIfAbrupt(argList). */
  @ReturnIfAbrupt(argList);
  /* 7. Let result be Construct(func, argList, newTarget). */
  result := Construct(scope, null, func, argList, newTarget);
  /* 8. ReturnIfAbrupt(result). */
  @ReturnIfAbrupt(result);
  /* 9. Let thisER be GetThisEnvironment( ). */
  thisER := GetThisEnvironment(scope);
  /* 10. Return thisER.BindThisValue(result).*/
  return BindThisValue(thisER, result)
}
```

Figure 4.12: The internal function in ECMARef6 that handles super as a function call

Overall, the `super` keyword in JavaScript offers a powerful mechanism for implementing inheritance, facilitating code reuse, and enabling customization of parent class behaviors within child classes. Let us now expand on how ECMARef6 implements both usages of `super`.

Super as a Function In Figure 4.12, we show the function that handles the interpretation of `super()`. The only parameter is `arguments`, and they are mapped to the variable `Arguments`. The first line retrieves the new target, which is the object that `new` was called with. The second line ensures that the new target is not undefined. Next, the interpreter calls the internal function `GetSuperConstructor` and assigns the return value to the `func` variable. We will discuss this function in more detail later. The interpreter then uses the `JS_Interpreter_Arguments` internal function to interpret the arguments passed to `super`. In line 7, the `Construct` internal function is used to invoke the constructor function from `func`. Finally, lines 9 and 10 ensure that the `this` environment has the correct values. Throughout the specification, there are calls to `"@ReturnIfAbrupt"` to check for errors along the way.

The `GetSuperConstructor` function plays a crucial role in retrieving the parent constructor. Let's examine the specifications of this function, which are displayed in Figure 4.13. In the first line of the function, the variable `envRec` is assigned to the Environment Record from which the `super` call was made. This ensures that the function operates within the appropriate context. Moving to the second line, an assertion is made to verify that `envRec` is indeed an Environment Record, ensuring the function's reliability. The third line assigns the current function to the variable `activeFunction`, representing an object similar to those depicted in yellow in Figure 4.9. Next, in line 4, the `super` constructor variable is assigned to the prototype of `activeFunction`. This prototype corresponds to the object pointed to by the `[[Prototype]]` internal slot. Lines 5 and 6 perform necessary checks on the `superConstructor` variable, ensuring the validity of the retrieved data. Finally, on line 7, the object corresponding to the `super` constructor is returned. This operation can only be performed within the child's constructor, as the `activeFunction` variable always points to the current constructor function. Thus, this logical flow

Runtime Semantics: GetSuperConstructor ( )

The abstract operation GetSuperConstructor performs the following steps:

1. Let *envRec* be GetThisEnvironment( ).
2. Assert: *envRec* is a function Environment Record.
3. Let *activeFunction* be *envRec*.[[FunctionObject]].
4. Let *superConstructor* be *activeFunction*.[[GetPrototypeOf]]().
5. ReturnIfAbrupt(*superConstructor*).
6. If IsConstructor(*superConstructor*) is **false**, throw a **TypeError** exception.
7. Return *superConstructor*.

Figure 4.13: The specification for the `GetSuperConstructor` internal function

GetSuperBase ()

1. Let *envRec* be the function Environment Record for which the method was invoked.
2. Let *home* be the value of *envRec*.[[HomeObject]].
3. If *home* has the value **undefined**, return **undefined**.
4. Assert: Type(*home*) is Object.
5. Return *home*.[[GetPrototypeOf]]().

Figure 4.14: The specification for the `GetSuperBase` internal function

ensures the successful execution of the function.

Super as an object When using `super` as an object, the mechanism employed is similar to the one used in the case of a `super` call. The specification of the internal function `GetSuperBase` is illustrated in Figure 4.14. This function is responsible for retrieving the object referenced when `super` is used as an object. In line 1, the variable `envRec` is assigned to the current environment record, although the standard does not provide specific details on how to access the environment record at that particular point. Moving on to line 2, the variable `home` is assigned to the `[[HomeObject]]` internal slot of that environment record. The `[[HomeObject]]` contains the Prototype object for the corresponding class, which is depicted in green in Figure 4.9. Lines 3 and 4 perform necessary checks on the `home` variable, ensuring its validity. Finally, in line 5, the prototype for the `home` variable is returned. This prototype represents the object from which we seek to access properties using `super`.

Chapter 5

Evaluation

This chapter presents the evaluation of the ECMARef6 interpreter. To assess the interpreter's compliance with the standard we used the ECMAScript official test suite, Test262. The following subsections will expand on the structure of Test262, discuss the test execution pipeline and present a comprehensive analysis of the testing results.

5.1 Test262

Test262 [9] is the official ECMAScript test suite, used to evaluate the compliance of JavaScript interpreters with the ECMAScript standard. It consists of a large collection of test cases designed to cover the language specification thoroughly. The purpose of Test262 is to ensure that JavaScript implementations conform to the standard. The test suite was first created in 2012 and has been regularly updated to reflect changes to the ECMAScript standard. The test suite is maintained by the ECMA Technical Committee 39, responsible for the development and maintenance of the ECMAScript standard. As of the latest version, Test262 contains over 35,000 test cases covering various language features and edge cases, making it a comprehensive tool for assessing ECMAScript compliance.

Tests are not the only feature of Test262. In addition to its tests, it also provides the harness, which is a collection of files with auxiliary functions used by Test262 tests; these functions can be divided into the three following classes:

- **Value comparison**: this class includes functions primarily used for comparing values, such as: `sameValue(v1, v2)` for checking if v1 has the same value as v2 or `notSameValue(v1, v2)` for checking if v1 has not the same value as v2;
- **Property helpers**: this class includes functions used for checking internal properties of objects, such as: `verifyNotEnumerable(v1, 'prop')` for checking if `prop` property from `v1` is not enumerable; `verifyNotWritable(v1, 'prop')` for checking if `prop` property from `v1` is not writable; `verifyConfigurable(v1, 'prop')` for checking if `prop` property from `v1` is configurable;
- **Test262-Specific Errors**: this class includes the error constructor functions used to create the runtime errors thrown by Test262, such as `assert.throws(expectedErrorConstructor, func)` to check if `func` will throw the error corresponding to `expectedErrorConstructor`.

In addition to utilizing harness functions, Test262 tests also make use of metadata to facilitate the organization and categorization of the test suite. This metadata contains relevant information about each test, such as the version of the ECMAScript standard to which it applies and the features that it

covers. The metadata is included in the tests in the form of code comments that consist of key-value pairs, where each key represents a test property. The most relevant keys are:

- **esid**: identifies the section of the ECMAScript specification that the test covers;
- **description**: provides a brief summary of the test case;
- **info**: contains additional information about the test case, such as the exact lines of the standard at which the test is aimed;
- **features**: lists the JavaScript features that the test is designed to test;
- **negative**: specifies whether the test is expected to throw an error or not;
- **includes**: a reference to a harness file that needs to be included for the test to work.

The metadata of Test262 tests serves as a valuable resource for developers and testers to understand the purpose and scope of each test case. It provides a standardized way of organizing and categorizing test cases, allowing for easier management and maintenance of Test262. Additionally, the metadata can be used to generate reports and metrics on test coverage and compliance with ECMAScript standards.

Importantly, some of the Test262 metadata keys have been deprecated but not updated in their corresponding tests. In particular, the keys `es5id` and `es6id`, which referenced the section of tests aimed at the version 5 and 6 of the standard, have been replaced by a new key `esid` described above. However, the majority of tests created for the 5th and 6th versions of the standard still contain the keys `es5id` and `es6id`.

Example Listing 11 presents a test262 test aimed at the testing the semantics of the property name of class objects. The first two lines include the copyright and author information, followed by the test metadata from lines 3 to 17. The metadata properties in this test are: es6id, description, info, includes and features. Lines 19-29 are the test itself with lines 23-29 being the assertion phase of the test. The harness functions used by the tests are: `sameValue`, `notSameValue`, `verifyNotEnumerable`, `verifyNotWritable` and `verifyConfigurable`

While, in principle, one could use the deprecated keys es6id and es5id, for identifying the tests applicable to our reference interpreter, this methodology is not precise because it ignores all the tests that target features of ES5 and ES5 and use the proper esid key, which does not expose the version. As a result, we had to manually select the tests, analyzing each one individually. This was necessary for a comprehensive evaluation of the interpreter, as the automatic selection of tests was too complex to be resolved within the scope of this thesis.

5.2 Test Selection

Selecting the appropriate tests from Test262 for the ES6 standard is a challenging task. While the `es5id` and `es6id` metadata properties can help in identifying the version of the standard to which a test belongs, they were deprecated in 2016 and are no longer sufficient for selecting tests. As a result, developers and testers must resort to alternative methods to identify relevant tests.

One solution to this problem is manual filtering. This involves executing all the tests present in Test262 and carefully analyzing the results of each test, if a test fails it can fall into one of two categories:

- **An ECMARef6 implementation bug:** the test is related to ES6 but the interpreter has a bug;

```
// Copyright (C) 2015 the V8 project authors. All rights reserved.
// This code is governed by the BSD license found in the LICENSE file.

/*---
es6id: 13.3.1.4
description: Assignment of function `name` attribute (ClassExpression)
info: |
    LexicalBinding : BindingIdentifier Initializer
    [...]
    6. If IsAnonymousFunctionDefinition(Initializer) is true, then
       a. Let hasNameProperty be HasOwnProperty(value, "name").
       b. ReturnIfAbrupt(hasNameProperty).
       c. If hasNameProperty is false, perform SetFunctionName(value,
          bindingId).
includes: [propertyHelper.js]
features: [class]
---*/

const xCls = class x {};
const cls = class {};
const xCls2 = class { static name() {} };

assert.notSameValue(xCls.name, 'xCls');
assert.notSameValue(xCls2.name, 'xCls2');

assert.sameValue(cls.name, 'cls');
verifyNotEnumerable(cls, 'name');
verifyNotWritable(cls, 'name');
verifyConfigurable(cls, 'name');
```

Listing 11: A test related to constant declarations and classes in ES6

- **A feature not supported by ES6:** the test uses a non-ES6 feature.

However, we did not have to manually filter all Test262 tests. Test262 is organised in folders, with each folder testing a specific ECMAScript feature. Hence, we started by ignoring all the folders specifically aimed at a feature introduced in a later version of the standard (7 onward). Although this process is time-consuming and complex, it is necessary for accurate evaluation of the interpreter's compliance with the ES6 standard.

Table 5.1 presents the results of the filtering process, which involved removing tests from the Test262 repository based on two criteria: folder removal and manual inspection. The process started with an earlier commit from 2016, which contained `23,643` tests. We then filtered `2,978` tests from this commit to arrive at our version of the tests, which contains `20,665` tests that were used to evaluate our reference interpreter.

In Table 5.2, we provide further insight into the selected tests by showing the number of tests that have the `es5id` key, the `es6id` key, and the `esid` key. We observe that the majority of tests have the `esid` key, while a smaller number of tests have the `es5id` and `es6id` keys. Moreover, most of the selected tests do not include the deprecated keys. This information helps us better understand the characteristics of the tests we used to evaluate our reference interpreter.

Initial number of tests	23643
Filtered number of tests	20665

Table 5.1: Filtered tests from test262

Tests with es5id	8482
Tests with es6id	2966
Tests with esid	9108

Table 5.2: Metadata analysis from filtered tests

Due to the difficulty of selecting appropriate tests for ECMARef6 compliance, we have kept the filtered version of the Test262 test suite in our repository along with our interpreter. This serves as a valuable resource for future development and testing of the interpreter, as well as for maintaining and improving its compliance with the ES6 standard.

5.3 Evaluation Pipeline

The execution of tests in the ECMA-SL ecosystem is a highly complex and challenging process that requires the integration of multiple systems. To provide a clearer understanding of the pipeline, we have decided to begin by illustrating the single test execution pipeline in detail. By explaining this process thoroughly, we can establish the complexities involved. After that, we can expand our focus and address optimizations and changes necessary for executing a batch of tests at once, with maximum efficiency.

Figure 5.1 illustrates the single test pipeline, and we will explain each highlighted point to provide a comprehensive understanding of the process:

1. First, we concatenate the test code and the harness code and use the JS2ECMA-SL tool to generate the corresponding code in ECMA-SL (`ast.esl`).
2. The entry point for the interpreter (`main.esl`) imports both the interpreter code and the generated ECMA-SL file, which includes the harness and the test code. This entry point executes the interpreter using `ast.esl` as an argument, effectively running the program.
3. The `main.esl` file is then converted to CoreECMA-SL using the ECMA-SL2CoreECMA-SL tool. The resulting file is `core.cesl`, which is the same program but in a smaller subset of the ECMA-SL language.
4. The test outcome can be categorized as follows: *Ok* (success), *Fail* (failure), *Error* (interpreter issue), or *Unsupported* (unimplemented/unsupported feature).

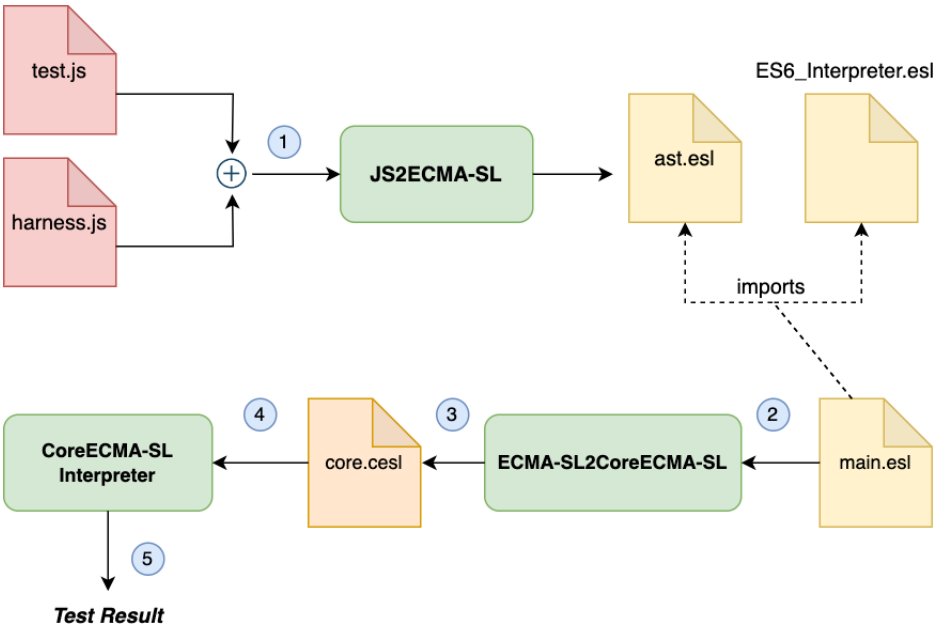


Figure 5.1: The execution pipeline for a single test from the Test262 suite

The execution of a single test is efficient, but when running a series of tests, certain parts of the process can become redundant. This is primarily due to the fact that the harness and interpreter code remain unchanged throughout the sequence. Therefore, there exists an opportunity for improvement when executing batches of tests. To optimize the process, we can implement changes such as caching the common elements of the test execution, reducing the time and effort required to execute the test sequence, and improving the overall performance of the testing process. The following are some potential strategies that can be employed to enhance the batch testing procedure:

1. Cache the interpreter code and harness in memory to avoid repeated recompilation and reloading.
2. Optimize the generation of CoreECMA-SL code for each test case to minimize the processing overhead.
3. Parallelize the execution of tests to further optimize the performance of the testing process.

While optimizing the execution of multiple tests is a crucial aspect of software testing, it is not sufficient on its own. The ECMARef project has multiple contributors, and to ensure the quality and stability of the project, it is essential to implement a Continuous Integration (CI) system. CI is a development practice that automates the testing and building of code changes in real-time. Our strategy for the ECMARef project was to implement a CI system that tracks the current state of tests locally, and provides a report on the performance of the tests after each code change. This way, contributors can be more confident that their changes to the interpreter will not negatively impact the overall performance of the project. By implementing this approach, the project can maintain a high level of quality and stability, while also enabling a collaborative development process.

5.4 Results

The results of the selected tests from Test262 are presented in Tables 5.3, 5.4, 5.5, and 5.6. To make it easier to navigate the large number of tests, we divided them into four tables based on the Test262 folder structure: language and built-ins. Since the focus of this thesis is on the language folder, we provide the results of the two most important sub-folders, expressions and statements, separately in Tables 5.3 and 5.4. Table 5.5, shows the results of the remaining sub-folders in the language folder, while Table 5.6 presents the tests from the built-ins folder.

Path	Ok	Fail	Error	Unsupported	Total	Success Percentage
language/expressions/array	11	0	0	0	**11**	100.00%
language/expressions/arrow-function	293	1	1	28	**323**	90.71%
language/expressions/assignment	291	45	0	30	**366**	79.51%
language/expressions/assignmenttargettype	7	0	0	0	**7**	100.00%
language/expressions/call	38	0	0	0	**38**	100.00%
language/expressions/class	455	0	0	0	**455**	100.00%
language/expressions/comma	5	0	0	0	**5**	100.00%
language/expressions/compound-assignment	408	22	0	0	**430**	94.88%
language/expressions/concatenation	5	0	0	0	**5**	100.00%
language/expressions/conditional	19	0	0	0	**19**	100.00%
language/expressions/delete	99	1	0	0	**100**	99.00%
language/expressions/function	21	0	0	0	**21**	100.00%
language/expressions/grouping	10	0	0	0	**10**	100.00%
language/expressions/in	14	0	0	0	**14**	100.00%
language/expressions/instanceof	43	0	0	0	**43**	100.00%
language/expressions/new	17	0	0	0	**17**	100.00%
language/expressions/object	53	0	0	0	**53**	100.00%
language/expressions/property-accessors	20	0	0	0	**20**	100.00%

language/expressions/super	66	3	0	2	**71**	92.96%
language/expressions/this	7	0	0	0	**7**	100.00%
language/expressions/typeof	15	0	0	0	**15**	100.00%
language/expressions/void	9	0	0	0	**9**	100.00%
language/expressions/operations	666	3	0	0	**669**	99.55%
language/expressions/postfix-and-prefix	126	8	0	0	**128**	98.43%
language/expressions/unary-operations	66	1	0	0	**67**	98.50%
Expressions	2773	84	1	60	**2918**	95.04%

Table 5.3: Test results for our subset of Test262

Path	Ok	Fail	Error	Unsupported	Total	Success Percentage
language/statements/async-function	1	0	0	0	**1**	100.00%
language/statements/block	14	5	0	0	**19**	73.68%
language/statements/break	20	0	0	0	**20**	100.00%
language/statements/class	533	8	0	1	**542**	98.34%
language/statements/const	105	2	19	10	**136**	77.21%
language/statements/continue	23	0	0	0	**23**	100.00%
language/statements/debugger	2	0	0	0	**2**	100.00%
language/statements/do-while	33	2	0	0	**35**	94.29%
language/statements/empty	2	0	0	0	**2**	100.00%
language/statements/expression	3	0	0	0	**3**	100.00%
language/statements/for-in	88	19	7	0	**114**	77.19%
language/statements/for-of	417	64	146	109	**736**	56.66%
language/statements/for	283	20	45	33	**381**	74.28%
language/statements/function	226	6	0	0	**232**	97.41%
language/statements/if	47	12	0	0	**59**	79.66%
language/statements/labeled	6	0	0	0	**6**	100.00%
language/statements/let	112	4	19	10	**145**	77.24%
language/statements/return	16	0	0	0	**16**	100.00%
language/statements/switch	42	66	0	0	**108**	38.89%
language/statements/throw	14	0	0	0	**14**	100.00%
language/statements/try	170	2	15	14	**201**	84.58%
language/statements/variable	169	12	7	10	**198**	85.35%
language/statements/while	32	5	0	0	**37**	86.49%
language/statements/with	166	5	0	0	**171**	97.08%
Statements	2524	232	258	187	**3201**	78.85%

Table 5.4: Test results for our subset of Test262

Path	Ok	Fail	Error	Unsupported	Total	Success Percentage
language/arguments-object	75	0	0	0	**75**	100.00%
language/ascii	101	0	0	0	**101**	100.00%
language/block-scope	8	0	0	0	**8**	100.00%
language/comments	19	0	0	0	**19**	100.00%
language/directive-prologue	57	5	0	0	**62**	91.94%
language/eval-code	57	0	0	0	**57**	100.00%
language/function-code	211	1	0	0	**212**	99.53%
language/future-reserved-words	54	0	0	0	**54**	100.00%
language/global-code	3	0	0	0	**3**	100.00%
language/identifier-resolution	12	0	0	0	**12**	100.00%
language/identifiers	88	0	0	0	**88**	100.00%
language/keywords	25	0	0	0	**25**	100.00%
language/line-terminators	34	7	0	0	**41**	82.93%
language/literals/regexp	136	15	0	0	**151**	90.07%
language/punctuators	11	0	0	0	**11**	100.00%
language/reserved-words	13	0	0	0	**13**	100.00%
language/rest-parameters	11	0	0	0	**11**	100.00%
language/source-text	0	1	0	0	**1**	0.00%
language/statementList	31	0	0	0	**31**	100.00%
language/types	109	0	0	0	**109**	100.00%
language/white-space	40	0	0	0	**40**	100.00%

Other Language Tests	1095	28	291	251	**1123**	97.50%

Table 5.5: Test results for the remainder sub-folders of the `langugage` folder

Path	Ok	Fail	Error	Unsupported	Total	Success Percentage
built-ins/Array	2677	20	1	1	**2701**	99.11%
built-ins/ArrayBuffer	78	10	1	0	**89**	87.64%
built-ins/ArrayIteratorPrototype	22	5	0	0	**27**	81.48%
built-ins/Boolean	51	0	0	0	**51**	100.00%
built-ins/DataView	323	12	0	0	**335**	96.42%
built-ins/Date	740	10	0	0	**750**	98.67%
built-ins/Error	41	0	0	0	**41**	100.00%
built-ins/Function	374	25	0	0	**399**	93.73%
built-ins/Infinity	7	0	0	0	**7**	100.00%
built-ins/IteratorPrototype	4	0	0	0	**4**	100.00%
built-ins/JSON	140	5	5	0	**150**	93.33%
built-ins/Map	154	2	0	0	**156**	98.72%
built-ins/MapIteratorPrototype	11	0	0	0	**11**	100.00%
built-ins/Math	337	4	0	0	**341**	98.83%
built-ins/NaN	7	0	0	0	**7**	100.00%
built-ins/NativeErrors	43	0	0	0	**43**	100.00%
built-ins/Number	303	45	0	0	**348**	87.07%
built-ins/Object	2941	1	0	0	**2942**	99.97%
built-ins/Promise	375	9	0	0	**384**	97.66%
built-ins/Proxy	252	7	0	1	**260**	96.92%
built-ins/Reflect	149	3	0	0	**152**	98.03%
built-ins/RegExp	883	508	19	0	**1410**	62.62%
built-ins/Set	196	1	0	0	**197**	99.49%
built-ins/SetIteratorPrototype	11	0	0	0	**11**	100.00%
built-ins/String	973	35	6	0	**1014**	95.96%
built-ins/StringIteratorPrototype	6	1	0	0	**7**	85.71%
built-ins/Symbol	67	14	0	0	**81**	82.72%
built-ins/TypedArray	624	1	0	0	**626**	99.68%
built-ins/TypedArrayConstructors	318	16	0	2	**336**	94.64%
built-ins/WeakMap	91	2	0	0	**93**	97.85%
built-ins/WeakSet	78	1	0	0	**79**	98.73%
built-ins/decodeURI	6	0	0	0	**6**	100.00%
built-ins/decodeURIComponent	6	0	0	0	**6**	100.00%
built-ins/encodeURI	6	0	0	0	**6**	100.00%
built-ins/encodeURIComponent	6	0	0	0	**6**	100.00%
Built-ins	12467	749	32	4	**13255**	94.06%

Table 5.6: Test results for built-ins part of Test262

The ECMARef6 project was subjected to a total of 20655 tests, with an overall success rate of 92.03%. Out of the total tests, 19009 were successful, while 1101 failed, 291 encountered errors, and 251 were unsupported. The tests are divided into two primary folders: built-ins and language. The built-ins folder had a success rate of 94.06%, with 12467 successful tests, 749 failures, 32 errors, and 4 unsupported tests out of a total of 13255 tests. The language folder had a success rate of 88.41%, with 6542 successful tests, 352 failures, 259 errors, and 247 unsupported tests out of a total of 7400 tests. The language folder consisted of two primary sub-folders, expressions, and statements, which accounted for most of the errors encountered. Other folders contain occasional errors, mostly related to edge cases that require careful analysis to be fixed.

The sub-folder that has encountered the most errors is the built-ins/regexp folder. Within this folder, there are 893 successful tests, 508 failed tests, and 19 tests with errors. The primary reason for these errors is due to Unicode. Unicode is a character encoding standard that enables the representation of a diverse range of characters, including non-Latin scripts like Chinese, Arabic, and Cyrillic. However, our interpreter currently does not fully support Unicode, which is the main cause of these errors.

To illustrate a case, consider Listing 12, which tests whether a Unicode character can be inserted into a regex query. The test begins on line 19 with a declaration of a string variable. Then, on line 21, an assert statement uses the match method from the string to search using a regex query. The regex query searches for a Unicode character that is not supported, causing the test to fail.

In addition to Unicode errors, we have other errors scattered throughout the test result tables. These errors can fall into two categories: an edge case that requires effort and causes the failure of a few tests, or an uncommon case that is thoroughly tested and causes the failure of several dozen tests. To fix these tests, they must be carefully investigated. The difficulty lies in the fact that it is often challenging to find the implementation detail that is related to the test, given that the interpreter is complex and has many execution layers.

```
1  // Copyright (C) 2020 Apple Inc. All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3  /*---
4  author: Michael Saboff
5  description: Exotic named group names in Unicode RegExps
6  esid: prod-GroupSpecifier
7  features: [regexp-named-groups]
8  ---*/
9
10 /*
11  Valid ID_Start / ID_Continue Unicode characters
12    \u{1d4d1}  \ud835 \udcd1
13    \u{1d4fb}  \ud835 \udcfb
14    \u{1d4f8}  \ud835 \udcf8
15    \u{1d500}  \ud835 \udd00
16    \u{1d4f7}  \ud835 \udcf7
17 */
18
19 var string = "The quick brown fox jumped over the lazy dog's back";
20
21 assert.sameValue(string.match(/(?<>brown)/u).groups., "brown");
```

Listing 12: A failing test related to unicode support

Chapter 6

Conclusions and Future Work

In this thesis, we implemented a reference interpreter for version 6 of the ECMAScript Standard. Our primary objective was to develop a robust and comprehensive interpreter that accurately captures the core aspects of the language. We dedicated extensive efforts to ensure the correctness and trustworthiness of our implementation. Throughout the development process, we placed great emphasis on testing and adhering to the standard specification. Rigorous testing procedures were employed to thoroughly evaluate the interpreter's functionality and compliance with the ECMAScript Standard. Additionally, we meticulously followed the guidelines outlined in the specification, striving to faithfully replicate the language's behavior in our implementation.

This resulted in ECMARef6, the most comprehensive academic reference interpreter for ES6 to date: it passes over 19K tests out of a total 35K tests for the most recent version of the standard, while JISET, the second most complete reference interpreter only passing 18K. The completeness and accuracy of ECMARef6 make it a valuable tool for researchers, developers, and educators alike.

Future Work Looking towards the future, our work holds potential for further development and expansion. ECMARef6 serves as a solid foundation that can be extended and built upon in various ways. Here are some key areas where future work can be directed to enhance and leverage our reference interpreter effectively:

- **Completing the ES6 implementation**: While our implementation of ES6 is nearly complete, we acknowledge the presence of minor bugs that require attention. Additionally, there are a few essential built-ins, including Modules, Generators, and Async Functions, that are yet to be fully implemented. Addressing these remaining tasks will ensure the comprehensive coverage and functionality of our interpreter;
- **Leveraging ECMARef6 to implement newer versions**: The robust core established in ECMARef6 positions it as an excellent foundation for implementing subsequent versions of the ECMAScript Standard. As newer versions primarily introduce new built-in libraries, our existing work greatly facilitates the implementation process for these additions.
- **New Projects**: In addition to the ongoing `ECMA-SL` project, our reference interpreter opens up numerous possibilities for the development of various tools for managing the ECMAScript standardisation process. For instance, the `ECMA-SL` project team is actively working on a tool to generate the HTML version of the standard directly from the `ECMA-SL` implementation of the interpreter. Furthermore, there are numerous other tools that can leverage the capabilities of our interpreter, such as:

- an automatic test suite coverage evaluator for assessing the coverage of existing JavaScript test suites, most notably Test262;
- an automatic test suite generator for automatically synthesising tests for new features of the language; and
- debuggers that allow developers to code-step not only the code of their programs but also the pseudo-code of the standard while interpreting their programs.

These are just a few examples of the many opportunities for innovative projects that can benefit from our reference interpreter.

Bibliography

[1] "Nodejs - an open-source, cross-platform javascript runtime environment." https://nodejs.org/en. Accessed on 2023-05-27.

[2] "Jerryscript - a javascript engine for internet of things." https://jerryscript.net/. Accessed on 2023-05-27.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] J. Park, J. Park, S. An, and S. Ryu, *JISET: JavaScript IR-Based Semantics Extraction Toolchain*, p. 647–658. Association for Computing Machinery, 2020.

[9] "Test262 - official ecmascript conformance test suite." https://github.com/tc39/test262. Accessed on 2023-03-14.

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

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

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

[13] "Ocaml - general-purpose, multi-paradigm programming language," 1996. Accessed on 2022-01-07.

[14] P. Thiemann, "Towards a type system for analyzing javascript programs," vol. 3444, 04 2005.

[15] C. Anderson, S. Drossopoulou, and P. Giannini, "Towards type inference for javascript," vol. 3586, 07 2005.

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

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

[18] K. Dewey, V. Kashyap, and B. Hardekopf, “A parallel abstract interpreter for javascript,” *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 34–45, 2015.

[19] A. Chaudhuri, “Flow: Abstract interpretation of javascript for type checking and beyond,” pp. 1–1, 10 2016.

[20] J. Fragoso Santos, T. Jensen, T. Rezk, and A. Schmitt, “Hybrid typing of secure information flow in a javascript-like language,” pp. 63–78, 01 2016.

[21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “Jsflow: Tracking information flow in javascript and its apis,” *Proceedings of the ACM Symposium on Applied Computing*, 03 2014.

[22] A. Chudnov and D. A. Naumann, “Inlined information flow monitoring for javascript,” *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

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

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

[25] “Coq - interactive formal proof management system.,” 1989. Accessed on 2022-01-08.

[26] “K - rewrite-based executable semantic framework.,” 2010. Accessed on 2022-01-08.

[27] S. Maffeis, J. C. Mitchell, and A. Taly, “An operational semantics for javascript,” in *APLAS*, 2008.

[28] S. Maffeis and A. Taly, “Language-based isolation of untrusted javascript,” in *2009 22nd IEEE Computer Security Foundations Symposium*, pp. 77–91, 2009.

[29] S. Maffeis, J. C. Mitchell, and A. Taly, “Isolating javascript with filters, rewriting, and wrappers,” in *ESORICS*, 2009.

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

[31] “Racket - general-purpose programming language.,” 1995. Accessed on 2022-01-08.

[32] P. Gibbs, J. CarrollMatthew, S. LernerBenjamin, PombrioJustin, and KrishnamurthiShriram, “A tested semantics for getters, setters, and eval in javascript,” *Sigplan Notices*, 2012.

[33] M. Bodin, T. Jensen, and A. Schmitt, “Pretty-big-step-semantics-based certified abstract interpretation,” *Electronic Proceedings in Theoretical Computer Science*, vol. 129, 09 2013.

[34] “V8 - google’s open source high-performance javascript and webassembly engine, written in c++.,” 2008. Accessed on 2022-01-08.

[35] G. Rosu, A. Stefanescu, S. Ciobaca, and B. Moore, “One-path reachability logic,” pp. 358–367, 06 2013.

[36] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, “Semantics-based program verifiers for all languages,” *ACM SIGPLAN Notices*, vol. 51, pp. 74–91, 10 2016.

[37] J. Launchbury and S. L. P. Jones, “State in haskell,” *LISP and Symbolic Computation*, vol. 8, pp. 293–341, 1995.

[38] G. Sampaio, J. F. Santos, P. Maksimovic, and P. Gardner, “A trusted infrastructure for symbolic analysis of event-driven web applications,” in *ECOOP*, 2020.