Inconsistencies in the ECMA-262 specification syntax

Alberto Tontoni

University of Bergen

Motivation

The current version of the ECMA-262 specification includes inconsistent (different) notations to express the same behavior.

In addition, some algorithm steps are too complex to be parsed by state-of-the-art language support tools.

My goal is to highlight these cases and suggest a suitable solution to address them, in order to increase the quality of the specification.

During my thesis work, I’ve found the following 3 (+1 extra) inconsistencies:

- Specifying how to access object field references
- Specifying if-then-else steps
- Algorithms represented as tables
- Steps that are too big (and complex) to be parsed by ESMeta

Object field references

Let *calleeRealm* be *F*.[[Realm]].

Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.

Let *methods* be the value of *constructor*.[[PrivateMethods]].

Let *description* be *name*'s [[Description]] value.

Object field references

Let *calleeRealm* be *F*.[[Realm]].

Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.

Let *methods* be the value of *constructor*.[[PrivateMethods]].

Let *description* be *name*'s [[Description]] value.

Since all notations imply the same semantics, I suggest updating the specification to use only one notation.

Object field references

Let *calleeRealm* be *F*.[[Realm]].

Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.

Let *methods* be the value of *constructor*.[[PrivateMethods]].

Let *description* be *name*'s [[Description]] value.

Since all notations imply the same semantics, I suggest updating the specification to use only one notation.

Object field references

Let *calleeRealm* be *F*.[[Realm]].

Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.

```
Set D.[[Value]] to X.[[Value]].
```

Let *methods* be the value of *constructor*.[[PrivateMethods]].

```
Let methods be constructor.[[PrivateMethods]].
```

Let *description* be *name*'s [[Description]] value.

```
Let description be name.[[Description]].
```

Since all notations imply the same semantics, I suggest updating the specification to use only one notation.

If-then-else

Then, else clauses in the same step:

1. If DaysInYear(YearFromTime(*t*)) is $\mathbf{366}_{\mathbb{F}}$, return $\mathbf{1}_{\mathbb{F}}$; else return $\mathbf{+0}_{\mathbb{F}}$.

Then, else clauses in two separate steps:

2. If *argument* is an abrupt completion, return Completion(*argument*).
3. Else, set *argument* to *argument*.[[Value]].

“otherwise” keyword (*without comma*):

c. If *min* = 0, let *min2* be 0; otherwise let *min2* be *min* - 1.

“otherwise” keyword (*with comma*):

a. If *x* and *y* are both **true** or both **false**, return **true**; otherwise, return **false**.

If-then-else

Then, else clauses in the same step:

1. If DaysInYear(YearFromTime(*t*)) is $\mathbf{366}_{\mathbb{F}}$, return $\mathbf{1}_{\mathbb{F}}$; else return $\mathbf{+0}_{\mathbb{F}}$.

Then, else clauses in two separate steps:

2. If *argument* is an abrupt completion, return Completion(*argument*).
3. Else, set *argument* to *argument*.[[Value]].

"otherwise" keyword (*without comma*):

c. If *min* = 0, let *min2* be 0; otherwise let *min2* be *min* - 1.

"otherwise" keyword (*with comma*):

a. If *x* and *y* are both **true** or both **false**, return **true**; otherwise, return **false**.

- Enforce the use of only one keyword between "else" and "otherwise"
- Standardize whether or not to use a comma after the "otherwise" keyword (or just replace both cases with an "else" keyword)
- Enforce a consistent style: either then, else always in the same step, or always in separate steps

Algorithms represented as tables

7.1.17 ToString (*argument*)

The abstract operation ToString takes argument *argument* (an ECMAScript language value) and returns either a normal completion containing a String or a throw completion. It converts *argument* to a value of type String. It performs the following steps when called:

1. If *argument* is a String, return *argument*.
2. If *argument* is a Symbol, throw a **TypeError** exception.
3. If *argument* is **undefined**, return **"undefined"**.
4. If *argument* is **null**, return **"null"**.
5. If *argument* is **true**, return **"true"**.
6. If *argument* is **false**, return **"false"**.
7. If *argument* is a Number, return Number::toString(*argument*, 10).
8. If *argument* is a BigInt, return BigInt::toString(*argument*, 10).
9. Assert: *argument* is an Object.
10. Let *primValue* be ? ToPrimitive(*argument*, STRING).
11. Assert: *primValue* is not an Object.
12. Return ? ToString(*primValue*).

Table 14: RequireObjectCoercible Results

Argument Type	Result
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return *argument*.
Number	Return *argument*.
String	Return *argument*.
Symbol	Return *argument*.
BigInt	Return *argument*.
Object	Return *argument*.

Table 13: ToObject Conversions

Argument Type	Result
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal slot is set to *argument*. See 20.3 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal slot is set to *argument*. See 21.1 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal slot is set to *argument*. See 22.1 for a description of String objects.
Symbol	Return a new Symbol object whose [[SymbolData]] internal slot is set to *argument*. See 20.4 for a description of Symbol objects.
BigInt	Return a new BigInt object whose [[BigIntData]] internal slot is set to *argument*. See 21.2 for a description of BigInt objects.
Object	Return *argument*.

Algorithms represented as tables

Not parsed by ESMeta

Table 14: RequireObjectCoercible Results

Argument Type	Result
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return *argument*.
Number	Return *argument*.
String	Return *argument*.
Symbol	Return *argument*.
BigInt	Return *argument*.
Object	Return *argument*.

Parsed by ESMeta

7.1.17 ToString (*argument*)

The abstract operation ToString takes argument *argument* (an ECMAScript language value) and returns either a normal completion containing a String or a throw completion. It converts *argument* to a value of type String. It performs the following steps when called:

1. If *argument* is a String, return *argument*.
2. If *argument* is a Symbol, throw a **TypeError** exception.
3. If *argument* is **undefined**, return **"undefined"**.
4. If *argument* is **null**, return **"null"**.
5. If *argument* is **true**, return **"true"**.
6. If *argument* is **false**, return **"false"**.
7. If *argument* is a Number, return Number::toString(*argument*, 10).
8. If *argument* is a BigInt, return BigInt::toString(*argument*, 10).
9. Assert: *argument* is an Object.
10. Let *primValue* be ? ToPrimitive(*argument*, STRING).
11. Assert: *primValue* is not an Object.
12. Return ? ToString(*primValue*).

Not parsed by ESMeta

Table 13: ToObject Conversions

Argument Type	Result
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal slot is set to *argument*. See 20.3 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal slot is set to *argument*. See 21.1 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal slot is set to *argument*. See 22.1 for a description of String objects.
Symbol	Return a new Symbol object whose [[SymbolData]] internal slot is set to *argument*. See 20.4 for a description of Symbol objects.
BigInt	Return a new BigInt object whose [[BigIntData]] internal slot is set to *argument*. See 21.2 for a description of BigInt objects.
Object	Return *argument*.

Extra - CreateIntrinsics

9.3.2 CreateIntrinsics (*realmRec*)

The abstract operation CreateIntrinsics takes argument *realmRec* (a Realm Record) and returns UNUSED. It performs the following steps when called:

1. Set *realmRec*.[[Intrinsics]] to a new Record.
2. Set fields of *realmRec*.[[Intrinsics]] with the values listed in Table 6. The field names are the names listed in column one of the table. The value of each field is a new object value fully and recursively populated with property values as defined by the specification of each object in clauses 19 through 28. All object property values are newly created object values. All values that are built-in function objects are created by performing CreateBuiltinFunction(*steps*, *length*, *name*, *slots*, *realmRec*, *prototype*) where *steps* is the definition of that function provided by this specification, *name* is the initial value of the function's **"name"** property, *length* is the initial value of the function's **"length"** property, *slots* is a list of the names, if any, of the function's specified internal slots, and *prototype* is the specified value of the function's [[Prototype]] internal slot. The creation of the intrinsics and their properties must be ordered to avoid any dependencies upon objects that have not yet been created.
3. Perform AddRestrictedFunctionProperties(*realmRec*.[[Intrinsics]].[[%Function.prototype%]], *realmRec*).
4. Return UNUSED.

- Step 2 is remarkably large and currently not parsed by ESMeta
- Break the step into a series of steps that are more easily parseable