Corrections to Sept. 1. ES5 Approval Draft

Sept. 22 Corrections

(Various minor typological changes were also made.)

7.2 White Space

White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space characters may occur between any two tokens, and at the start or end of input. White space characters and may also occur within a *StringLiteral* or a *RegularExpressionLiteral* (where they are considered significant characters forming part of the literal value) or within a *Comment*, but cannot appear within any other kind of token.

7.3 Line Terminators

… Any lLine terminators may only occur within a *StringLiteral* token as part of a *LineContinuation*must be preceded by an escape sequence.

11.8.5 The Abstract Relational Comparison Algorithm

NOTE 2 The comparison of Strings uses a simple lexicographic ordering on sequences of code unit values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore String values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalised form. Also, note that for strings containing supplementary characters, lexicographic ordering on sequences of UTF-16 code unit values differs from that on sequences of code point values.

14.1 Directive Prologues and the Use Strict Directive

A Directive Prologue is the longesta sequence of zero or more *ExpressionStatement* productions, each of which having an *Expression* consists entirely of a *StringLiteral*, that occuroccurring as the initial *SourceElement* productions of a *Program* or *FunctionBody*. and where each *ExpressionStatement* in the sequence consists entirely of a *StringLiteral* token followed a semicolon. The semicolon may appear explicitly or may be inserted by automatic semicolon insertion. A Directive Prologue may be an empty sequence.

15.1.3 URI Handling Function Properties

When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is first transformed into a sequence of octets using the UTF-8 transformation, with surrogate pairs first transformed from their UTF-16 to UTF-32 encodings. (Note that for code units in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a String with each octet represented by an escape sequence of the form "%xx".

15.2.3.7 Object.defineProperties (O, Properties)

6. Atomically, forFor each element *desc* of *descriptors* in list order,

~~Step 6 of the above algorithm is specified as a sequential loop in which calls to [[DefineOwnProperty]] may throw an exception at various intermediate points. Rather than failing after a partial update of *O*, this step must be implemented such that it either atomically completes all property updates successfully or fails without making any update to the properties of object *O*.~~

15.10.6.2 RegExp.prototype.exec(string)

4. Let *lastIndex* be the result of calling the [[Get]] internal method of *R* with argument "**`lastIndex`**".

11. If *global* is **true**,

Sept. 21 Corrections

10.1.1 Strict Mode Code

An ECMAScript *Program* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as strict global code, strict eval code, and strict function code. Code is interpreted ~~in~~ as strict mode code in the following situations:

- Global code is strict global code if it begins with a Directive Prologue that contains ~~the *Program* that defines the global code includes~~ a Use Strict Directive (see 14.1).

- Eval code is strict eval code if it begins with a Directive Prologue that contains ~~is a *Program* that includes~~ a Use Strict Directive or if the call to eval is a direct call (see 15.1.2.1.1) to the eval function that is contained in strict mode code.

- Function code that is part of a *FunctionDeclaration* ~~or~~, *FunctionExpression*, or accessor *PropertyAssignment* is strict function code if its *FunctionDeclaration*, ~~or~~ *FunctionExpression*, or *PropertyAssignment* is contained in strict mode code or if the function code begins with a Directive Prologue that contains~~its *FunctionBody* includes~~ a Use Strict Directive.

- Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a String that when processed as a *FunctionBody* begins with a Directive Prologue that contains~~includes~~ a Use Strict Directive.

10.2.1.1.5 Delete~~t~~Binding (N)

10.2.1.2.5 Delete~~t~~Binding (N)

14 Program

1. The code of this *Program* is strict mode code ~~if the Directive Prologue (14.1) of its *SourceElements* contains a Use Strict Directive or~~ if any of the conditions of 10.1.1 apply. If the code of this *Program* is strict mode code, *SourceElements* is evaluated in the following steps as strict mode code. Otherwise *SourceElements* is evaluated in the following steps as non-strict mode code.

15.3.2.1 new Function (p1, p2, … , pn, body)

8. If *body* is not parsable as *FunctionBody* then throw a **SyntaxError** exception.
~~9. *body* is strict mode code if when it is parsed as a *FunctionBody* the first *SourceElement* within *SourceElements* is a *Statement* production that is an *ExpressionStatement* whose *Expression* consists solely of a *StringLiteral* whose value is a Use Strict Directive (14.1).~~
~~10.~~9. If *body* is strict mode code (see 10.1.1) then let *strict* be **true**, else let *strict* be **false**.
~~11.~~10. If *strict* is **true**, throw any exceptions specified in 13.1 that apply.
~~12.~~11. Return a new Function object created as specified in 13.2 passing *P* as the *FormalParameterList* and *body* as the *FunctionBody*. Pass in the Global Environment as the *Scope* parameter and *strict* as the *Strict* flag.

15.10.6.2 RegExp.prototype.exec(string)

~~4.~~12. Call the [[DefineOwnProperty]] internal method of *A* with arguments "**`length`**", Property Descriptor {[[Value]]: *n* + 1~~, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**~~}, and **true**.

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, … + Start at: 11 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

Annex D

(added)

15.3.4.3: In Edition 3, a **TypeError** is thrown if the second argument passed to **`Function.prototype.apply`** is neither an array object nor an arguments object. In Edition 5, the second argument may be any kind of generic array-like object that has a valid **`length`** property.

Sept. 17 Corrections

11.9.6 The Strict Equality Comparison Algorithm

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
~~a.~~4. If Type(*x*) is Number, then
    ~~b.~~a. If *x* is **NaN**, return **false**.
    ~~c.~~b. If *y* is **NaN**, return **false**.
    ~~d.~~c. If *x* is the same Number value as *y*, return **true**.
    ~~e.~~d. If *x* is **+0** and *y* is **–0**, return **true**.
    ~~f.~~e. If *x* is **–0** and *y* is **+0**, return **true**.
    ~~g.~~f. Return **false**.
~~4.~~5. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
~~5.~~6. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
~~6.~~7. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.

Formatted

7 Lexical Conventions

~~As a workaround, one may enclose the regular expression literal in parentheses.~~

8.12.9 [[DefineOwnProperty]] (P, Desc, Throw)

In the following algorithm, the term "Reject" means "If *Throw* is **true**, then throw a **TypeError** exception, otherwise return **false**~~.~~". The algorithm contains steps that test various fields of the Property Descriptor *Desc* for specific values. The fields that are tested in this manner need not actually exist in *Desc*. If a field is absent then its value is considered to be **false**.

(step 7)

b. Reject, if the [[Enumerable]] field of *Desc* is present and ~~if~~ the [[Enumerable]] fields of *current* and *Desc* are the Boolean negation of each other.

~~NOTE 1 The above algorithm contains steps that test various fields of the Property Descriptor *Desc* for specific values. The fields that are tested in this manner need not actually exist in *Desc*. If a field is absent then the result of any test of its value is logically false.~~

NOTE ~~2~~ Step 10.b allows any field of Desc to be different from the corresponding field of current if current's [[Configurable]] field is **true**. This even permits changing the [[Value]] of a property whose [[Writable]] attribute is **false**. This is allowed because a **true** [[Configurable]] attribute would permit an equivalent sequence of calls where [[Writable]] is first set to **true**, a new [[Value]] is set, and then [[Writable]] is set to **false**.

10.6 Arguments Object

The [[Get]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

The [[GetOwnProperty]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

The [[DefineOwnProperty]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P*, Property Descriptor *Desc*, and Boolean flag *Throw* performs the following steps:

The [[Delete]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* and Boolean flag *Throw* performs the following steps:

NOTE 1 ~~Array~~ For non-strict mode functions the array index (defined in 15.4) named data properties of an arguments object whose numeric name values are less than the number of formal parameters of the corresponding function object initially share their values with the corresponding argument bindings in the function's execution context. This means that changing the property changes the corresponding value of the argument binding and vice-versa. This correspondence is broken if such a property is deleted and then redefined or if the property is changed into an accessor property. For strict mode functions, the values of the arguments object's properties are simply a copy of the arguments passed to the function and there is no dynamic linkage between the property values and the formal parameter values.

15.3.4.5 Function.prototype.bind (thisArg [, arg1 [, arg2, …]])

NOTE Function objects created using **`Function.prototype.bind`** do not have a **`prototype`** property or a the [[Code]], [[FormalParameters]], and [[Scope]] internal ~~property~~properties.

15.5.5 Properties of String Instances

String instances inherit properties from the String prototype object and their [[Class]] internal property value is **"String"**. String instances also have a [[PrimitiveValue]] internal property, a **length** property, and a set of enumerable properties with array index names.

15.6.5 Properties of Boolean Instances

Boolean instances inherit properties from the Boolean prototype object and their [[Class]] internal property value is **"Boolean"**. Boolean instances also have a [[PrimitiveValue]] internal property.

15.7.5 Properties of Number Instances

Number instances inherit properties from the Number prototype object and their [[Class]] internal property value is **"Number"**. Number instances also have a [[PrimitiveValue]] internal property.

15.9.6 Properties of Date Instances

Date instances inherit properties from the Date prototype object and their [[Class]] internal property value is **"Date"**. Date instances also have a [[PrimitiveValue]] internal property.

15.10.2.16 NonemptyClassRangesNoDash

NOTE 3 ~~Even~~ A **-** character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of *ClassRanges*, the beginning or end limit of a range specification, or immediately follows a range specification.

15.10.7 Properties of RegExp Instances

RegExp instances inherit properties from the RegExp prototype object and their [[Class]] internal property value is **"RegExp"**. RegExp instances also have a [[Match]] internal property and a **length** property.

15.12.1.2 The JSON Syntactic Grammar

The JSON Syntactic Grammar defines a valid JSON text in terms of tokens defined by the JSON lexical grammar. The goal symbol of the grammar is ~~*JSONValue*~~*JSONText*.

Annex D

(added)

15.5.5.2: In Edition 5, the individual characters of a String object's [[PrimitiveValue] may be accessed as array indexed properties of the String object. These properties are non-writable and non-configurable and shadow any inherited properties with the same names. In Edition 3, these properties did not exist and ECMAScript code could dynamically add and remove writable properties with such names and could access inherited properties with such names.

Sept. 12 Corrections

(reverted Sept. 17)

~~8.12.9 [[DefineOwnProperty]] (P, Desc, Throw)~~

~~NOTE 1 The above algorithm contains steps that test various fields of the Property Descriptor *Desc* for specific values. The fields that are tested in this manner need not actually exist in *Desc*. If a field is absent then its value is considered to be **false**. result of any test of its value is logically false.~~

10.6 Arguments Object

(MakeArgGetter)

2. Return the result of creating a function object as described in 13.2 using no *FormalParameterList*, *body* for *FunctionBody*, ~~and~~ *env* as *Scope*, and **true** for *Strict*.

(MakeArgSetter)

3. Return the result of creating a function object as described in 13.2 using a List containing the single String *param* as *FormalParameterList*, *body* for *FunctionBody*, ~~and~~ *env* as *Scope*, and **true** for *Strict*.

~~The **toString** method of an arguments object performs the following steps when called with a this value *O* and an argument list *L*:~~

~~1. Let *toString* be the result of calling the [[Get]] internal method of the standard built-in object **Object.prototype** passing **"toString"** as the argument.~~
~~2. If IsCallable(*toString*) is **false**, throw a **TypeError** exception.~~
~~3. Return the result of calling the [[Call]] internal method of *toString* passing *O* as the **this** value and *L* as the arguments list.~~

~~The **toLocaleString** method of an arguments object performs the following steps when called with a this value *O* and an argument list *L*:~~

~~1. Let *toLocaleString* be the result of calling the [[Get]] internal method of the standard builtin object **Object.prototype** passing **"toLocaleString"** as the argument.~~
~~2. If IsCallable(*toString*) is **false**, throw a **TypeError** exception.~~
~~3. Return the result of calling the [[Call]] internal method of *toLocaleString* passing *O* as the **this** value and *L* as the arguments list.~~

~~NOTE These definitions of **toString** and **toLocaleString** exist to provide compatibility with Edition 3 where argument objects inherit from **Object.prototype** instead of **Array.prototype**.~~

11.1.5 Object Initialiser

(production *PropertyAssignment* : **get** *PropertyName* **( ) {** *FunctionBody* **}**)

2. Let *closure* be the result of creating a new Function object as specified in 13.2 with an empty parameter list and body specified by *FunctionBody*. Pass in the LexicalEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *PropertyAssignment* is contained in strict code or if its *FunctionBody* is strict code.

(production *PropertyAssignment* : **set** *PropertyName* **(** *PropertySetParameterList* **) {** FunctionBody **}**)

2. Let *closure* be the result of creating a new Function object as specified in 13.2 with parameters specified by *PropertySetParameterList* and body specified by *FunctionBody*. Pass in the LexicalEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *PropertyAssignment* is contained in strict code or if its *FunctionBody* is strict code.

It is a **SyntaxError** if the *Identifier* **"eval"** or the *Identifier* **"arguments"** occurs as the *Identifier* in a *PropertySetParameterList* of a *PropertyAssignment* that is contained in strict code or if its *FunctionBody* is strict code.

The production *PropertyName* **:** *IdentifierName* is evaluated as follows:

Annex C

(add bullet item between existing 11.1.5 and 10.4.2 items)

- It is a **SyntaxError** if the *Identifier* **"eval"** or the *Identifier* **"arguments"** occurs as the *Identifier* in a *PropertySetParameterList* of a *PropertyAssignment* that is contained in strict code or if its *FunctionBody* is strict code (11.1.5).

Sept. 10 Corrections

8.7.1 GetValue (V)

1. If Type(*V*) is not Reference, return *V*.
2. Let *base* be the result of calling GetBase(*V*).
3. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsPropertyReference(*V*), then
   a. If HasPrimitiveBase(*V*) is **false**, then let *get* be the [[Get]] internal method of *base*, otherwise let *get* be the special [[Get]] internal method defined below. ~~If HasPrimitiveBase(*V*), then let *base* be ToObject(*base*) (see 9.9).~~
   b. Return the result of calling the ~~[[Get]]~~*get* internal method ~~of *base*~~using *base* as its **this** value, and~~,~~ passing GetReferencedName(*V*) for the argument.
5. Else, *base* must be an environment record.
   a. Return the result of calling the GetBindingValue (see 10.2.1) concrete method of *base* passing GetReferencedName(*V*) and IsStrictReference(*V*) as arguments.

The following [[Get]] internal method is used by GetValue when *V* is a property reference with a primitive base value. It is called using *base* as its **this** value and with property *P* as its argument. The following steps are taken:

1. Let *O* be ToObject(*base*).
2. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
3. If *desc* is **undefined**, return **undefined**.
4. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
5. Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]].
6. If *getter* is **undefined**, return **undefined**.
7. Return the result calling the [[Call]] internal method of *getter* providing *base* as the **this** value and providing no arguments.

NOTE The object that may be created in step 1 is not accessible outside of the above method.~~step 4.a is immediately discarded after its use in the next step.~~ An implementation might choose to avoid the actual creation of the object. The only situation where such an actual property access that uses this internal method can have visible effect is when it invokes an accessor function.

8.7.2 PutValue (V, W)

(Step 4)

b. Call the *put* internal method using *base* as its **this** ~~object~~value, and passing GetReferencedName(*V*) for the property name, *W* for the value, and IsStrictReference(*V*) for the *Throw* flag.

(Special [[Put]] internal method, step 6)

b. Call the [[Call]] internal method of *setter* providing ~~*Base*~~ *base* as the **this** value and an argument list containing only *W*.

Sept. 8 Corrections

7.8.4 String Literals

(last paragraph before NOTE)

A conforming implementation, when processing strict mode code (see 10.1.1), may not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.~~1~~2.

11.13.1 Simple Assignment (=)

NOTE When an assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable reference. If it does a **ReferenceError** exception is thrown upon assignment. The *LeftHandSide* also may not be a reference to a data property with the attribute value {[[Writable]]:**false**}, to an accessor property with the attribute value {[[Set~~Put~~]]:**undefined**}, nor to a non-existent property of an object whose [[Extensible]] internal property has the value **false**. In these cases a **TypeError** exception is thrown.

15.2.3.4 Object.getOwnPropertyNames (O)

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *array* be the result of creating a new object as if by the expression **`new Array ()`** where **`Array`** is the standard built-in constructor with that name.
3. Let *n* be 0.
4. For each named own property *P* of *O*
   ~~5.~~a. Let *name* be the String value that is the name of *P*.
   ~~6.~~b. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*n*), the PropertyDescriptor {[[Value]]: *name*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
   ~~7.~~c. Increment *n* by 1.
~~8.~~5. Return *array*.

Formatted

15.4.4.14 Array.prototype.indexOf (searchElement [, fromIndex])

`indexOf` compares *searchElement* to the elements of the array, in ascending order, using the internal Strict Equality Comparison Algorithm ~~SameValue comparison operation~~ (11.9.6~~9.12~~), and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

9. b

i. Let *same* be the result of applying the Strict Equality Comparis~~i~~on Algorithm to *searchElement* and *elementK*.

15.4.4.15 Array.prototype.lastIndexOf (searchElement [, fromIndex])

`lastIndexOf` compares *searchElement* to the elements of the array in descending order using the internal Strict Equality Comparison Algorithm (11.9.6)~~SameValue comparison operation (9.12)~~, and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

8. b

ii. Let *same* be the result of applying the Strict Equality Comparis~~i~~on Algorithm to *searchElement* and *elementK*.

Annex C

(add as second bullet item)

- A conforming implementation, when processing strict mode code (see 10.1.1), may not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.2.

(original second bullet)

- Assignment to an undeclared identifier or otherwise unresolvable reference does not create a property in the global object. When a simple assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable Reference. If it does a **ReferenceError** exception is thrown (8.7.2). The *LeftHandSide* also may not be a reference to a data property with the attribute value {[[Writable]]:**false**}, to an accessor property with the attribute value {[[~~Put~~Set]]:**undefined**}, nor to a non-existent property of an object whose [[Extensible]] internal property has the value **false**. In these cases a **`TypeError`** exception is thrown (11.13.1).

Annex ~~D~~E

(added)

15.3.5.2: In Edition 5, the **`prototype`** property of Function instances is not enumerable. In Edition 3, this property was enumerable.