NOTE: I am using colours in this document to ensure that character styles are applied consistently. They can be removed by changing Word's character styles and will be removed for the final draft.

Sections 5 through 5.12.1 and all of chapters 9 and 10 are new or revised to the point where having change bars in them would be pointless. The change bars are accurate in the other parts of this document.

1 Scope

This Standard defines the ECMAScript Edition 4 scripting language.

2 Conformance

3 Normative References

4 Overview

5 Notational Conventions

This specification uses the notation below to represent algorithms and concepts. These concepts are used as notation only and are not necessarily represented or visible in the ECMAScript language.

5.1 Text

Throughout this document, the phrase *code point* and the word *character* is used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of Unicode text in the UTF-16 transformation format. The phrase *Unicode character* is used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code point). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual Unicode characters, even though a user might think of the whole sequence as a single character.

When denoted in this specification, characters with values between 20 and 7E hexadecimal inclusive are in a `fixed width font`. Other characters are denoted by enclosing their four-digit hexadecimal Unicode value between «u and ». For example, the non-breakable space character would be denoted in this document as «u00A0». A few of the common control characters are represented by name:

Abbreviation	Unicode Value
«NUL»	«u0000»
«BS»	«u0008»
«TAB»	«u0009»
«LF»	«u000A»
«VT»	«u000B»
«FF»	«u000C»
«CR»	«u000D»
«SP»	«u0020»

A space character is denoted in this document either by a blank space where it's obvious from the context or by «SP» where the space might be confused with some other notation.

5.2 Tags

Tags are computational tokens with no internal structure. Tags are written using a **bold sans-serif font**. Two tags are equal if and only if they have the same name. Examples of tags include **true**, **false**, **null**, **NaN**, and **identifier**.

5.3 Booleans

The tags **true** and **false** represent *booleans*. BOOLEAN is the two-element set {**true**, **false**}.

Let *a* and *b* be booleans. In addition to = and ≠, the following operations can be done on them:

not *a*	**true** if *a* is **false**; **false** if *a* is **true**
a **and** *b*	If *a* is **false**, returns **false** without computing *b*; if *a* is **true**, returns the value of *b*
a **or** *b*	If *a* is **false**, returns the value of *b*; if *a* is **true**, returns **true** without computing *b*
a **xor** *b*	**true** if *a* is **true** and *b* is **false** or *a* is **false** and *b* is **true**; **false** otherwise. *a* **xor** *b* is equivalent to $a \neq b$

Note that the **and** and **or** operators short-circuit. These are the only operators that do not always compute all of their operands.

5.4 Sets

A set is an unordered, possibly infinite collection of elements. Each element may occur at most once in a set. There must be an equivalence relation = defined on all pairs of the set's elements. Elements of a set may themselves be sets.

A set is denoted by enclosing a comma-separated list of values inside braces:

$\{element_1, element_2, \ldots, element_n\}$

The empty set is written as {}. Any duplicate elements are included only once in the set.

For example, the set {3, 0, 10, 11, 12, 13, -5} contains seven integers.

Sets of either integers or characters can be abbreviated using the ... range operator. For example, the above set can also be written as {0, –5, 3 ... 3, 10 ... 13}.

If the beginning of the range is equal to the end of the range, then the range consists of only one element: {7 ... 7} is the same as {7}. If the end of the range is one less than the beginning, then the range contains no elements: {7 ... 6} is the same as {}. The end of the range is never more than one less than the beginning.

A set can also be written using the set comprehension notation

$\{f(x) \mid \forall x \in A;\ predicate_1(x); \ldots ;\ predicate_n(x)\}$

which denotes the set of the results of computing expression *f* on all elements *x* of set *A* that simultaneously satisfy all *predicate* expressions. There can also be more than one free variable *x*, in which case all combinations of free variables' values are considered. For example,

$\{x \mid \forall x \in \text{INTEGER};\ x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}$

$\{x \times 10 + y \mid \forall x \in \{1, 2, 4\};\ \forall y \in \{3, 5\}\} = \{13, 15, 23, 25, 43, 45\}$

Let *A* and *B* be sets and *x* and *y* be values. The following notation is used on sets:

$x \in A$	**true** if *x* is an element of set *A* and **false** if not
$x, y \in A$	**true** if *x* and *y* are both elements of set *A* and **false** if not

$x \notin A$	**false** if x is an element of set A and **true** if not
$\|A\|$	The number of elements in the set A (only used on finite sets)
min A	The value m that satisfies both $m \in A$ and for all elements $x \in A$, $x \geq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
max A	The value m that satisfies both $m \in A$ and for all elements $x \in A$, $x \leq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
$A \cap B$	The intersection of sets A and B (the set of all values that are present both in A and in B)
$A \cup B$	The union of sets A and B (the set of all values that are present in at least one of A or B)
$A - B$	The difference of sets A and B (the set of all values that are present in A but not B)
$A = B$	**true** if sets A and B are equal and **false** otherwise. sets A and B are equal if every element of A is also in B and every element of B is also in A.
$A \neq B$	**false** if the sets A and B are equal and **true** otherwise
$A \subseteq B$	**true** if A is a subset of B and **false** otherwise. A is a subset of B if every element of A is also in B. Every set is a subset of itself. The empty set {} is a subset of every set.
$A \subset B$	**true** if A is a proper subset of B and **false** otherwise. $A \subset B$ is equivalent to $A \subseteq B$ **and** $A \neq B$.
A{}	The *powerset* of A, which is the set of all subsets of A. For example, if A = {1,2,3}, then A{} = {{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}.

5.4.1 Constraint Sets

Sets are useful to describe the possible values that a variable might take on in an algorithm. The algorithms are constructed in a way that ensures that, in the absence of language extensions, these constraints are always met, regardless of any valid or invalid programmer or user input or actions. An implementation's language extensions may invalidate these constraints.

Sets used for constraints have names in CAPITALIZED RED SMALL CAPS. Such a name is to be considered distinct from a tag or regular variable with the same name, so UNDEFINED, **undefined**, and *undefined* are three different and independent things.

A variable v is constrained using the notation

v: T

where T is a set. This constraint indicates that the value of v will always be a member of the set T. These declarations are informative (they may be dropped without affecting the algorithms' correctness) but useful in understanding the algorithms.

5.5 Real Numbers

Numbers written in this specification are to be understood to be exact mathematical real numbers, which include integers and rational numbers as subsets. Examples of numbers include -3, 0, 17, 10^{1000}, and π. Hexadecimal numbers are written by preceding them with "0x", so 4294967296, 0x100000000, and 2^{32} are all the same integer.

INTEGER is the set of all integers {... –3, –2, –1, 0, 1, 2, 3 ...}. 3.0, 3, 0xFF, and -10^{100} are all integers.

RATIONAL is the set of all rational numbers. Every integer is also a rational number: INTEGER ⊂ RATIONAL. 3, 1/3, 7.5, –12/7, and 2^{-5} are examples of rational numbers.

REAL is the set of all real numbers. Every rational number is also a real number: RATIONAL ⊂ REAL. π is an example of a real number slightly larger than 3.14.

Let x and y be real numbers. The following operations can be done on them and always produce exact results:

$-x$	Negation
$x + y$	Sum
$x - y$	Difference
$x \times y$	Product
x / y	Quotient (y must not be zero)
x^y	x raised to the y^{th} power (used only when either $x \neq 0$ and y is an integer or x is any number and $y > 0$)
$\lvert x \rvert$	The absolute value of x, which is x if $x \geq 0$ and $-x$ otherwise
$\lfloor x \rfloor$	*Floor* of x, which is the unique integer i such that $i \leq x < i+1$. $\lfloor \pi \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, and $\lfloor 7 \rfloor = 7$.
$\lceil x \rceil$	*Ceiling* of x, which is the unique integer i such that $i-1 < x \leq i$. $\lceil \pi \rceil = 4$, $\lceil -3.5 \rceil = -3$, and $\lceil 7 \rceil = 7$.
x **mod** y	x modulo y, which is defined as $x - y \times \lfloor x/y \rfloor$. y must not be zero. 10 **mod** 7 = 3, and –1 **mod** 7 = 6.

Real numbers can be compared using =, ≠, <, ≤, >, and ≥. The result is either **true** or **false**. Multiple relational operators can be cascaded, so $x < y < z$ is **true** only if both x is less than y and y is less than z.

5.5.1 Bitwise Integer Operators

The four functions below perform bitwise operations on integers. The integers are treated as though they were written in infinite-precision two's complement binary notation, with each 1 bit representing **true** and 0 bit representing **false**. Zero or a positive integer is interpreted as having infinitely many consecutive 0's as its most significant bits, while a negative integer is interpreted as having infinitely many consecutive 1's as its most significant bits. For example, 6 is interpreted as ...0...0000110, while –6 is interpreted as ...1...1111010; ANDing them together yields ...0...0000010, which is the integer 2.

bitwiseAnd(*x*: INTEGER, *y*: INTEGER): INTEGER	The bitwise AND of x and y
bitwiseOr(*x*: INTEGER, *y*: INTEGER): INTEGER	The bitwise OR of x and y
bitwiseXor(*x*: INTEGER, *y*: INTEGER): INTEGER	The bitwise XOR of x and y
bitwiseShift(*x*: INTEGER, *count*: INTEGER): INTEGER	Shift x to the left by *count* bits. If *count* is negative, shift x to the right by –*count* bits. Bits shifted out of the right end are lost; bit shifted in at the right end are zero. *bitwiseShift*(*x*, *count*) is exactly equivalent to $\lfloor x \times 2^{count} \rfloor$.

5.6 Floating-Point Numbers

The set FLOAT64 denotes all representable double-precision floating-point IEEE 754 values, with all not-a-number values considered indistinguishable from each other. The set FLOAT64 is the union of the following sets:

FLOAT64 = NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64 ∪ {**+zero**, **-zero**, **+∞**, **-∞**, **NaN**}

There are 18428729675200069632 (that is, $2^{64}-2^{54}$) normalised values:

NORMALISEDFLOAT64 = $\{s{*}m{*}2^e \mid s \in \{-1, 1\};\ m, e \in$ INTEGER$;\ 2^{52} \leq m < 2^{53};\ -1074 \leq e \leq 971\}$

m is called the *significand*.

There are also 9007199254740990 (that is, $2^{53}–2$) denormalised non-zero values:

$\text{DENORMALISEDFLOAT64} = \{s*m*2^{-1074} \mid s \in \{-1, 1\}; m \in \text{INTEGER}; 0 < m < 2^{52}\}$

m is called the *significand*.

The remaining values are the tags **+zero** (positive zero), **-zero** (negative zero), **+∞** (positive infinity), **-∞** (negative infinity), and **NaN** (not a number).

The function *realToFloat64* converts a real number *x* into the applicable element of FLOAT64 as follows:

realToFloat64(*x*)

Let $S = \text{NORMALISEDFLOAT64} \cup \text{DENORMALISEDFLOAT64} \cup \{0, 2^{1024}, -2^{1024}\}$.

Let *a* be the element of *S* closest to *x* (i.e. such that $|a–x|$ is as small as possible). If two elements of *S* are equally close, let *a* be the one with an even significand; for this purpose 0, 2^{1024}, and -2^{1024} are considered to have even significands.

If $a = 2^{1024}$, return **+∞**.

If $a = -2^{1024}$, return **-∞**.

If $a \neq 0$, return *a*.

If $x < 0$, return **-zero**.

Return **+zero**.

NOTE This procedure corresponds exactly to the behaviour of the IEEE 754 "round to nearest" mode.

5.7 Characters

Characters enclosed in single quotes ‘ and ’ represent single Unicode 16-bit code points. Examples of characters include ‘A’, ‘b’, ‘«LF»’, and ‘«uFFFF»’ (see also section 5.1). Unicode surrogates are considered to be pairs of characters for the purpose of this specification.

CHARACTER is the set of all 65536 characters {‘«u0000»’ ... ‘«uFFFF»’}.

Characters can be compared using =, ≠, <, ≤, >, and ≥. These operators compare code point values, so ‘A’ = ‘A’, ‘A’ < ‘B’, and ‘A’ < ‘a’ are all **true**.

5.8 Lists

A finite ordered list of zero or more elements is written by listing the elements inside bold brackets:

[$element_0$, $element_1$, ... , $element_{n–1}$**]**

For example, the following list contains four strings:

[“parsley”, “sage”, “rosemary”, “thyme”**]**

The empty list is written as **[]**.

Unlike a set, the elements of a list are indexed by integers starting from 0. A list can contain duplicate elements.

Let u = **[**e_0, e_1, ... , $e_{n–1}$**]** and v = **[**f_0, f_1, ... , $f_{m–1}$**]** be lists, *i* and *j* be integers, and *x* be a value. The operations below can be done on lists. The operations are meaningful only when their preconditions are met; the semantics never use the operations below without meeting their preconditions.

Notation	Precondition	Description
$\lvert u\rvert$		The length *n* of the list
$u[i]$	$0 \le i < \lvert u\rvert$	The i^{th} element e_i.

$u[i \ldots j]$	$0 \le i \le j+1 \le \lvert u\rvert$	The list slice $[e_i, e_{i+1}, \ldots, e_j]$ consisting of all elements of u between the i^{th} and the j^{th}, inclusive. The result is the empty list [] if $j=i-1$.
$u[i \ldots]$	$0 \le i \le \lvert u\rvert$	The list slice $[e_i, e_{i+1}, \ldots, e_{n-1}]$ consisting of all elements of u between the i^{th} and the end. The result is the empty list [] if $i=n$.
$u[i \setminus x]$	$0 \le i < \lvert u\rvert$	The list $[e_0, \ldots, e_{i-1}, x, e_{i+1}, \ldots, e_{n-1}]$ with the i^{th} element replaced by the value x and the other elements unchanged
$[g(a) \mid \forall a \in u]$		The list $[g(e_0), g(e_1), \ldots, g(e_{n-1})]$ whose elements consist of the results of applying expression g to each element of u. a is the name of the parameter in expression g.
$[g(a) \mid \forall a \in u$ and $c(a)]$		Same as $[g(a) \mid \forall a \in u]$ except that the boolean expression c is first applied to each element of u. Only the elements for which c returns **true** are included in the result.
$u \oplus v$		The concatenated list $[e_0, e_1, \ldots, e_{n-1}, f_0, f_1, \ldots, f_{m-1}]$
$u = v$		**true** if the lists u and v are equal and **false** otherwise. Lists u and v are equal if they have the same length and all of their corresponding elements are equal.
$u \ne v$		**false** if the lists u and v are equal and **true** otherwise.

If T is a set, then T[] is the set of all lists whose elements are members of T. The empty list **[]** is a member of T[] for any set T.

5.9 Strings

A list of characters is called a *string*. In addition to the normal list notation, for notational convenience a string can also be written as zero or more characters enclosed in double quotes (see also the notation for non-ASCII characters). Thus,

“`Wonder`«LF»”

is equivalent to:

[‘`W`’, ‘`o`’, ‘`n`’, ‘`d`’, ‘`e`’, ‘`r`’, ‘«LF»’**]**

The empty string is usually written as “”.

In addition to all of the other list operations, <, ≤, >, and ≥ are defined on strings. A string x is less than string y when y is not the empty string and either x is the empty string, the first character of x is less than the first character of y, or the first character of x is equal to the first character of y and the rest of string x is less than the rest of string y.

STRING is the set of all strings. STRING = CHARACTER[].

5.10 Tuples

A *tuple* is an immutable aggregate of values comprised of a tag (section 5.2) and zero or more labelled fields.

The fields of each kind of tuple used in this specification are described in tables such as:

Field	Contents	Note
label$_1$	T_1	Informative note about this field
…	…	…

label*n*	T_N	Informative note about this field

label$_1$ through **label**$_n$ are the names of the fields. T_1 through T_N are informative sets of possible values that the corresponding fields may hold.

The notation

name$\langle v_1, \ldots, v_n\rangle$

represents a tuple with tag **name** and values v_1 through v_n for fields labelled **label**$_1$ through **label**$_n$ respectively. Each value v_i is a member of the corresponding set T_i.

If a is the tuple **name**$\langle v_1, \ldots, v_n\rangle$, then

a.**label**$_i$

returns the i^{th} field's value v_i.

The equality operators = and ≠ may be used to compare tuples. Tuples are equal when they have the same tag and their corresponding fields' values are equal.

5.11 Records

A *record* is a mutable aggregate of values similar to a tuple but with different equality behaviour.

A record is comprised of a tag (section 5.2) and an *address*. The address points to a mutable data structure comprised of zero or more labelled fields. The address acts as the record's serial number — every record allocated by **new** (see below) gets a different address, including records created by identical expressions or even the same expression used twice.

The fields of each kind of record used in this specification are described in tables such as:

Field	Contents	Note
label$_1$	T_1	Informative note about this field
...	...	...
label*n*	T_N	Informative note about this field

label$_1$ through **label**$_n$ are the names of the fields. T_1 through T_N are informative sets of possible values that the corresponding fields may hold.

The expression

new name$\langle\langle v_1, \ldots, v_n\rangle\rangle$

creates a record with tag **name** and a new address α. The fields labelled **label**$_1$ through **label**$_n$ at address α are initialised with values v_1 through v_n respectively. Each value v_i is a member of the corresponding set T_i.

If a is a record with tag **name** and address α, then

a.**label**$_i$

returns the current value v of the i^{th} field at address α. That field may be set to a new value w, which must be a member of the set T_i, using the assignment

a.**label**$_i \leftarrow w$

after which a.**label**$_i$ will evaluate to w. Any record with a different address β is unaffected by the assignment.

The equality operators = and ≠ may be used to compare records. Records are equal only when they have the same address.

5.12 Algorithm Steps

Steps of algorithms are described using a mixture of English and formal notation. The various kinds of steps are described in this section. Multiple steps are separated by semicolons or periods and performed in order unless an earlier step exits via a **return** or propagates an exception.

expression

A computation step may consist of an expression. The expression is computed and its value, if any, ignored.

v: T ← *expression*

v ← *expression*

An assignment step is indicated using the assignment operator ←. This step computes the value of *expression* and assigns the result to the temporary variable *v*. If this is the first time the variable is referenced in a function, the variable's constraint set T is listed; any value stored in *v* is guaranteed to be a member of the set T.

Temporary variables are local to the functions that define them (including any nested functions). Each time a function is called it gets a new set of temporary variables.

a.**label** ← *expression*

This form of assignment sets the value of field **label** of record *a* to the value of *expression*.

if $expression_1$ **then** *step*; *step*; ...; *step*
elsif $expression_2$ **then** *step*; *step*; ...; *step*
...
elsif $expression_n$ **then** *step*; *step*; ...; *step*
else *step*; *step*; ...; *step*
end if

An **if** step computes $expression_1$, which will evaluate to either **true** or **false**. If it is **true**, the first list of *step*s is performed. Otherwise, $expression_2$ is computed and tested, and so on. If no *expression* evaluates to **true**, the list of *step*s following the **else** is performed. The **else** clause may be omitted, in which case no action is taken when no *expression* evaluates to **true**.

case *expression* **of**
  set_1 **do** *step*; *step*; ...; *step*;
  set_2 **do** *step*; *step*; ...; *step*;
  ...;
  set_n **do** *step*; *step*; ...; *step*
  else *step*; *step*; ...; *step*
end case

A **case** step computes *expression*, which will evaluate to a value *v*. If $v \in set_1$, then the first list of *step*s is performed. Otherwise, if $v \in set_2$, then the second list of *step*s is performed, and so on. If *v* is not a member of any *set*, the list of *step*s following the **else** is performed. The **else** clause may be omitted, in which case *v* will always be a member of some *set*.

while *expression* **do**
  step; *step*; ...; *step*
end while

A **while** step computes *expression*, which will evaluate to either **true** or **false**. If it is **false**, no action is taken. If it is **true**, the list of *step*s is performed and then *expression* is computed and tested again. This repeats until *expression* returns **true** (or until the function exits via a **return** or an exception is propagated out).

return *expression*

A **return** step computes *expression* to obtain a value *v* and returns from the enclosing function with the result *v*. No further steps in the enclosing function are performed. The *expression* may be omitted, in which case the enclosing function returns with no result.

invariant *expression*

An **invariant** step is an informative note that states that computing *expression* at this point will always produce the value **true**.

throw *expression*

A **throw** step computes *expression* to obtain a value *v* and begins propagating exception *v* outwards, exiting partially performed steps and function calls until the exception is caught by a **catch** step. Unless the enclosing function catches this exception, no further steps in the enclosing function are performed.

try
  step; *step*; …; *step*
catch *v*: *set* **do**
  step; *step*; …; *step*
end try

A **try** step performs the first list of *step*s. If they complete normally (or if they **return** out of the current function), then the **try** step is done. If any of the *step*s propagates out an exception *e*, then if $e \in set$, then exception *e* stops propagating, variable *v* is bound to the value *e*, and the second list of *step*s is performed. If $e \notin set$, then exception *e* keeps propagating out.

A **try** step does not intercept exceptions that may be propagated out of its second list of *step*s.

5.12.1 Nested Functions

An inner **function** may be nested as a step inside an outer **function**. In this case the inner function is a closure and can access the parameters and temporaries of the outer function.

5.13 Grammars

The lexical and syntactic structure of ECMAScript programs is described in terms of *context-free grammars*. A context-free grammar consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet. A *grammar symbol* is either a terminal or a nonterminal.

Each grammar contains at least one distinguished nonterminal called the *goal symbol*. If there is more than one goal symbol, the grammar specifies which one is to be used. A *sentential form* is a possibly empty sequence of grammar symbols that satisfies the following recursive constraints:

- The sequence consisting of only the goal symbol is a sentential form.
- Given any sentential form α that contains a nonterminal *N*, one may replace an occurrence of *N* in α with the right-hand side of any production for which *N* is the left-hand side. The resulting sequence of grammar symbols is also a sentential form.

A *derivation* is a record, usually expressed as a tree, of which production was applied to expand each intermediate nonterminal to obtain a sentential form starting from the goal symbol. The grammars in this document are unambiguous, so each sentential form has exactly one derivation.

A *sentence* is a sentential form that contains only terminals. A *sentence prefix* is any prefix of a sentence, including the empty prefix consisting of no terminals and the complete prefix consisting of the entire sentence.

A *language* is the (perhaps infinite) set of a grammar's sentences.

5.13.1 Grammar Notation

Terminal symbols are either literal characters (section 5.1), sequences of literal characters (syntactic grammar only), or other terminals such as **Identifier** defined by the grammar. These other terminals are denoted in **bold**.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a ⇒ and one or more expansions of the nonterminal separated by vertical bars (|). The expansions are usually listed on separate lines but may be listed on the same line if they are short. An empty expansion is denoted as «empty».

To aid in reading the grammar, some rules contain informative cross-references to sections where nonterminals used in the rule are defined. These cross-references appear in parentheses in the right margin.

For example, the syntactic definition

SampleList ⇒
  «empty»
| `...` *Identifier* (*Identifier*: **Error! Reference source not found.**)
| *SampleListPrefix*
| *SampleListPrefix* `,` `...` *Identifier*

states that the nonterminal *SampleList* can represent one of four kinds of sequences of input tokens:

- It can represent nothing (indicated by the «empty» alternative).
- It can represent the terminal `...` followed by any expansion of the nonterminal *Identifier*.
- It can represent any expansion of the nonterminal *SampleListPrefix*.
- It can represent any expansion of the nonterminal *SampleListPrefix* followed by the terminals `,` and `...` and any expansion of the nonterminal *Identifier*.

5.13.2 Lookahead Constraints

If the phrase “[lookahead ∉ *set*]” appears in the expansion of a nonterminal, it indicates that that expansion may not be used if the immediately following terminal is a member of the given *set*. That *set* can be written as a list of terminals enclosed in curly braces. For convenience, *set* can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand.

For example, given the rules

DecimalDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

DecimalDigits ⇒
  DecimalDigit
| *DecimalDigits DecimalDigit*

the rule

LookaheadExample ⇒
  `n` [lookahead ∉ {`1`, `3`, `5`, `7`, `9`}] *DecimalDigits*
| *DecimalDigit* [lookahead ∉ {*DecimalDigit*}]

matches either the letter `n` followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

5.13.3 Line Break Constraints

If the phrase “[no line break]” appears in the expansion of a production, it indicates that this production cannot be used if there is a line break in the input stream at the indicated position. Line break constraints are only present in the syntactic grammar. For example, the rule

ReturnStatement ⇒
  `return`
| **`return`** [no line break] *ListExpression*$^{\text{allowIn}}$

indicates that the second production may not be used if a line break occurs in the program between the **return** token and the *ListExpression*$^{\text{allowIn}}$.

Unless the presence of a line break is forbidden by a constraint, any number of line breaks may occur between any two consecutive terminals in the input to the syntactic grammar without affecting the syntactic acceptability of the program.

5.13.4 Parameterised Rules

Many rules in the grammars occur in groups of analogous rules. Rather than list them individually, these groups have been summarised using the shorthand illustrated by the example below:

Metadefinitions such as

$\alpha \in$ {normal, initial}

$\beta \in$ {allowIn, noIn}

introduce grammar arguments α and β. If these arguments later parameterise the nonterminal on the left side of a rule, that rule is implicitly replicated into a set of rules in each of which a grammar argument is consistently substituted by one of its variants. For example, the sample rule

AssignmentExpression$^{\alpha,\beta}$ ⇒
ConditionalExpression$^{\alpha,\beta}$
| *LeftSideExpression*$^{\alpha}$ **=** *AssignmentExpression*$^{\text{normal},\beta}$
| *LeftSideExpression*$^{\alpha}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal},\beta}$

expands into the following four rules:

AssignmentExpression$^{\text{normal,allowIn}}$ ⇒
ConditionalExpression$^{\text{normal,allowIn}}$
| *LeftSideExpression*$^{\text{normal}}$ **=** *AssignmentExpression*$^{\text{normal,allowIn}}$
| *LeftSideExpression*$^{\text{normal}}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal,allowIn}}$

AssignmentExpression$^{\text{normal,noIn}}$ ⇒
ConditionalExpression$^{\text{normal,noIn}}$
| *LeftSideExpression*$^{\text{normal}}$ **=** *AssignmentExpression*$^{\text{normal,noIn}}$
| *LeftSideExpression*$^{\text{normal}}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal,noIn}}$

AssignmentExpression$^{\text{initial,allowIn}}$ ⇒
ConditionalExpression$^{\text{initial,allowIn}}$
| *LeftSideExpression*$^{\text{initial}}$ **=** *AssignmentExpression*$^{\text{normal,allowIn}}$
| *LeftSideExpression*$^{\text{initial}}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal,allowIn}}$

AssignmentExpression$^{\text{initial,noIn}}$ ⇒
ConditionalExpression$^{\text{initial,noIn}}$
| *LeftSideExpression*$^{\text{initial}}$ **=** *AssignmentExpression*$^{\text{normal,noIn}}$
| *LeftSideExpression*$^{\text{initial}}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal,noIn}}$

AssignmentExpression$^{\text{normal,allowIn}}$ is now an unparametrised nonterminal and processed normally by the grammar.

Some of the expanded rules (such as the fourth one in the example above) may be unreachable from the grammar's starting nonterminal; these are ignored.

5.13.5 Special Lexical Rules

A few lexical rules have too many expansions to be practically listed. These are specified by descriptive text instead of a list of expansions after the ⇒.

Some lexical rules contain the metaword **except**. These rules match any expansion that is listed before the **except** but that does not match any expansion after the **except**; if multiple expansions are listed after the **except**, then they are separated by vertical bars (|). All of these rules ultimately expand into single characters. For example, the rule below matches any single *UnicodeCharacter* except the `*` and `/` characters:

NonAsteriskOrSlash ⇒ *UnicodeCharacter* **except** `*` | `/`

6 Source Text

ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 2.1 or later, using the UTF-16 transformation format. The text is expected to have been normalised to Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

ECMAScript source text can contain any of the Unicode characters. All Unicode white space characters are treated as white space, and all Unicode line/paragraph separators are treated as line separators. Non-Latin Unicode characters are allowed in identifiers, string literals, regular expression literals and comments.

In string literals, regular expression literals and identifiers, any character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely **`\u`** plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

NOTE Although this document sometimes refers to a “transformation” between a “character” within a “string” and the 16-bit unsigned integer that is the UTF-16 encoding of that character, there is actually no transformation because a “character” within a “string” is actually represented using that 16-bit unsigned value.

NOTE ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence **`\u000A`**, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character **`000A`** is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence **`\u000A`** occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write **`\n`** instead of **`\u000A`** to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

6.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category `Cf` in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a Unicode escape sequence (see section **Error! Reference source not found.**) to include a Unicode format-control character inside a string or regular expression literal.

7 Lexical Grammar

This section defines ECMAScript's *lexical grammar*. This grammar translates the source text into a sequence of *input elements*, which are either tokens or the special markers **lineBreak** and **endOfInput**.

A *token* is one of the following:

- A **keyword** token, which is either:
  - One of the reserved words `abstract`, `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `enum`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `goto`, `if`, `implements`, `import`, `in`, `instanceof`, `interface`, `is`, `namespace`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `typeof`, `use`, `var`, `void`, `volatile`, `while`, `with`.
  - One of the non-reserved words `exclude`, `get`, `include`, `set`.
- A **punctuator** token, which is one of `!`, `!=`, `!==`, `#`, `%`, `%=`, `&`, `&&`, `&&=`, `&=`, `(`, `)`, `*`, `*=`, `+`, `++`, `+=`, `,`, `-`, `--`, `-=`, `->`, `.`, `..`, `...`, `/`, `/=`, `:`, `::`, `;`, `<`, `<<`, `<<=`, `<=`, `=`, `==`, `===`, `>`, `>=`, `>>`, `>>=`, `>>>`, `>>>=`, `?`, `@`, `[`, `]`, `^`, `^=`, `^^`, `^^=`, `{`, `|`, `|=`, `||`, `||=`, `}`, `~`.
- An **identifier** token, which carries a string that is the identifier's name.
- A **number** token, which carries a number that is the string's value.
- A **string** token, which carries a string that is the string's value.
- A **regularExpression** token, which carries two strings — the regular expression's body and its flags.

A **lineBreak**, although not considered to be a token, also becomes part of the stream of input elements and guides the process of automatic semicolon insertion (section **Error! Reference source not found.**). **endOfInput** signals the end of the source text.

NOTE The lexical grammar discards simple white space and single-line comments. They do not appear in the stream of input elements for the syntactic grammar. Comments spanning several lines become **lineBreak**s.

The lexical grammar has individual characters as its terminal symbols plus the special terminal **End**, which is appended after the last input character. The lexical grammar defines three goal symbols $\mathit{NextInputElement}^{\mathrm{re}}$, $\mathit{NextInputElement}^{\mathrm{div}}$, and $\mathit{NextInputElement}^{\mathrm{unit}}$, a set of productions, and instructions for translating the source text into input elements. The choice of the goal symbol depends on the syntactic grammar, which means that lexical and syntactic analysis are interleaved.

NOTE The grammar uses $\mathit{NextInputElement}^{\mathrm{unit}}$ if the previous token was a number, $\mathit{NextInputElement}^{\mathrm{re}}$ if the previous token was not a number and a `/` should be interpreted as starting a regular expression, and $\mathit{NextInputElement}^{\mathrm{div}}$ if the previous token was not a number and a `/` should be interpreted as a division or division-assignment operator.

The sequence of input elements *inputElements* is obtained as follows:

Let *inputElements* be an empty sequence of input elements.

Let *input* be the input sequence of characters. Append a special placeholder **End** to the end of *input*.

Let *state* be a variable that holds one of the constants **re**, **div**, or **unit**. Initialise it to **re**.

Repeat the following steps until exited:

Find the longest possible prefix *P* of *input* that is a member of the lexical grammar's language (see section 5.13). Use the start symbol *NextInputElement*$^{\text{re}}$, *NextInputElement*$^{\text{div}}$, or *NextInputElement*$^{\text{unit}}$ depending on whether *state* is **re**, **div**, or **unit**, respectively. If the parse failed, signal a syntax error.

Compute the action *Lex* on the derivation of *P* to obtain an input element *e*.

If *e* is **endOfInput**, then exit the repeat loop.

Remove the prefix *P* from *input*, leaving only the yet-unprocessed suffix of *input*.

Append *e* to the end of the *inputElements* sequence.

If the *inputElements* sequence does not form a valid sentence prefix of the language defined by the syntactic grammar, then:

If *e* is not **lineBreak**, but the next-to-last element of *inputElements* is **lineBreak**, then insert a **VirtualSemicolon** terminal between the next-to-last element and *e* in *inputElements*.

If *inputElements* still does not form a valid sentence prefix of the language defined by the syntactic grammar, signal a syntax error.

End if

If *e* is a **Number** token, then set *state* to **unit**. Otherwise, if the *inputElements* sequence followed by the terminal **/** forms a valid sentence prefix of the language defined by the syntactic grammar, then set *state* to **div**; otherwise, set *state* to **re**.

End repeat

If the *inputElements* sequence does not form a valid sentence of the context-free language defined by the syntactic grammar, signal a syntax error and stop.

Return *inputElements*.

7.1 Input Elements

Syntax

NextInputElement$^{\text{re}}$ ⇒ *WhiteSpace InputElement*$^{\text{re}}$ (*WhiteSpace*: 7.2)

NextInputElement$^{\text{div}}$ ⇒ *WhiteSpace InputElement*$^{\text{div}}$

NextInputElement$^{\text{unit}}$ ⇒
  [lookahead∉ {*ContinuingIdentifierCharacter*, ****}] *WhiteSpace InputElement*$^{\text{div}}$
| [lookahead∉ {**_**}] *IdentifierName* (*IdentifierName*: 7.5)
| **_** *IdentifierName*

InputElement$^{\text{re}}$ ⇒
  LineBreaks (*LineBreaks*: 7.3)
| *IdentifierOrKeyword* (*IdentifierOrKeyword*: 7.5)
| *Punctuator* (*Punctuator*: 7.6)
| *NumericLiteral* (*NumericLiteral*: 7.7)
| *StringLiteral* (*StringLiteral*: 7.8)
| *RegExpLiteral* (*RegExpLiteral*: 7.9)
| *EndOfInput*

InputElement$^{\text{div}}$ ⇒
  LineBreaks
| *IdentifierOrKeyword*
| *Punctuator*
| *DivisionPunctuator* (*DivisionPunctuator*: 7.6)
| *NumericLiteral*
| *StringLiteral*
| *EndOfInput*

EndOfInput ⇒
  End
| *LineComment* **End** (*LineComment*: 7.4)

Semantics

The grammar parameter *ν* can be either re or div.

Lex[*NextInputElement*$^{\text{re}}$ ⇒ *WhiteSpace InputElement*$^{\text{re}}$] = *Lex*[*InputElement*$^{\text{re}}$]

Lex[*NextInputElement*$^{\text{div}}$ ⇒ *WhiteSpace InputElement*$^{\text{div}}$] = *Lex*[*InputElement*$^{\text{div}}$]

Lex[*NextInputElement*$^{\text{unit}}$ ⇒ [lookahead∉ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace* *InputElement*$^{\text{div}}$] = *Lex*[*InputElement*$^{\text{div}}$]

Lex[*NextInputElement*$^{\text{unit}}$ ⇒ [lookahead∉ {_}] *IdentifierName*]
~~*Lex*[*NextInputElement*$^{\text{unit}}$ ⇒ _ *IdentifierName*]~~
Return a **string** token with string contents *LexString*[*IdentifierName*].

Lex[*InputElement*$^{\nu}$ ⇒ *LineBreaks*] = **lineBreak**

Lex[*InputElement*$^{\nu}$ ⇒ *IdentifierOrKeyword*] = *Lex*[*IdentifierOrKeyword*]

Lex[*InputElement*$^{\nu}$ ⇒ *Punctuator*] = *Lex*[*Punctuator*]

Lex[*InputElement*$^{\text{div}}$ ⇒ *DivisionPunctuator*] = *Lex*[*DivisionPunctuator*]

Lex[*InputElement*$^{\nu}$ ⇒ *NumericLiteral*] = *Lex*[*NumericLiteral*]

Lex[*InputElement*$^{\nu}$ ⇒ *StringLiteral*] = *Lex*[*StringLiteral*]

Lex[*InputElement*$^{\text{re}}$ ⇒ *RegExpLiteral*] = *Lex*[*RegExpLiteral*]

Lex[*InputElement*$^{\nu}$ ⇒ *EndOfInput*] = **endOfInput**

7.2 White space

Syntax

WhiteSpace ⇒
  «empty»
| *WhiteSpace WhiteSpaceCharacter*
| *WhiteSpace SingleLineBlockComment* (*SingleLineBlockComment*: 7.4)

WhiteSpaceCharacter ⇒
  «TAB» | «VT» | «FF» | «SP» | «u00A0»
| Any other character in category Zs in the Unicode Character Database

NOTE White space characters are used to improve source text readability and to separate tokens from each other, but are otherwise insignificant. White space may occur between any two tokens except between a number and an unquoted unit.

7.3 Line Breaks

Syntax

LineBreak ⇒
  LineTerminator
| *LineComment LineTerminator* (*LineComment*: 7.4)
| *MultiLineBlockComment* (*MultiLineBlockComment*: 7.4)

LineBreaks ⇒
  LineBreak
| *LineBreaks WhiteSpace LineBreak* (*WhiteSpace*: 7.2)

LineTerminator ⇒ «LF» | «CR» | «u2028» | «u2029»

NOTE Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section **Error! Reference source not found.**).

7.4 Comments

Syntax

LineComment ⇒ `/ /` *LineCommentCharacters*

LineCommentCharacters ⇒
  «empty»
| *LineCommentCharacters NonTerminator*

SingleLineBlockComment ⇒ `/ *` *BlockCommentCharacters* `* /`

BlockCommentCharacters ⇒
  «empty»
| *BlockCommentCharacters NonTerminatorOrSlash*
| *PreSlashCharacters* `/`

PreSlashCharacters ⇒
  «empty»
| *BlockCommentCharacters NonTerminatorOrAsteriskOrSlash*
| *PreSlashCharacters* `/`

MultiLineBlockComment ⇒ `/ *` *MultiLineBlockCommentCharacters BlockCommentCharacters* `* /`

MultiLineBlockCommentCharacters ⇒
  BlockCommentCharacters LineTerminator (*LineTerminator*: 7.3)
| *MultiLineBlockCommentCharacters BlockCommentCharacters LineTerminator*

UnicodeCharacter ⇒ Any character

NonTerminator ⇒ *UnicodeCharacter* **except** *LineTerminator*

NonTerminatorOrSlash ⇒ *NonTerminator* **except** `/`

NonTerminatorOrAsteriskOrSlash ⇒ *NonTerminator* **except** `*` | `/`

NOTE Comments can be either line comments or block comments. Line comments start with a `//` and continue to the end of the line. Block comments start with `/*` and end with `*/`. Block comments can span multiple lines but cannot nest.

Except when it is on the last line of input, a line comment is always followed by a *LineTerminator*. That *LineTerminator* is not considered to be part of that line comment; it is recognised separately and becomes a **lineBreak**. A block comment that actually spans more than one line is also considered to be a **lineBreak**.

7.5 Keywords and Identifiers

Syntax

IdentifierOrKeyword ⇒ *IdentifierName*

IdentifierName ⇒
  InitialIdentifierCharacterOrEscape
| *NullEscapes InitialIdentifierCharacterOrEscape*
| *IdentifierName ContinuingIdentifierCharacterOrEscape*
| *IdentifierName NullEscape*

Semantics

Lex[*IdentifierOrKeyword* ⇒ *IdentifierName*]

Let *id* be the string *LexString*[*IdentifierName*].

If *IdentifierName* contains no escape sequences (i.e. expansions of the *NullEscape* or *HexEscape* nonterminals) and exactly matches one of the keywords `abstract`, `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `enum`, `exclude`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `get`, `goto`, `if`, `implements`, `import`, `in`, `include`, `instanceof`, `interface`, `is`, `namespace`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `set`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `typeof`, `use`, `var`, `void`, `volatile`, `while`, `with`, then return a **keyword** token with string contents *id*.

Return an **identifier** token with string contents *id*.

NOTE Even though the lexical grammar treats `exclude`, `get`, `include`, and `set` as keywords, the syntactic grammar contains productions that permit them to be used as identifier names. The other keywords are reserved and may not be used as identifier names. However, an *IdentifierName* can never be a keyword if it contains any escape characters, so, for example, one can use `new` as the name of an identifier by including an escape sequence in it; `\_new` is one possibility, and `n\x65w` is another.

LexString[*IdentifierName* ⇒ *InitialIdentifierCharacterOrEscape*]

LexString[*IdentifierName* ⇒ *NullEscapes InitialIdentifierCharacterOrEscape*]

Return a one-character string with the character *LexChar*[*InitialIdentifierCharacterOrEscape*].

LexString[*IdentifierName* ⇒ $IdentifierName_1$ *ContinuingIdentifierCharacterOrEscape*]

Return a string consisting of the string *LexString*[$IdentifierName_1$] concatenated with the character *LexChar*[*ContinuingIdentifierCharacterOrEscape*].

LexString[*IdentifierName* ⇒ $IdentifierName_1$ *NullEscape*]

Return the string *LexString*[$IdentifierName_1$].

Syntax

NullEscapes ⇒
  NullEscape
| *NullEscapes NullEscape*

NullEscape ⇒ `\ _`

InitialIdentifierCharacterOrEscape ⇒
  InitialIdentifierCharacter
| `\` *HexEscape* (*HexEscape*: 7.8)

InitialIdentifierCharacter ⇒ *UnicodeInitialAlphabetic* | `$` | `_`

UnicodeInitialAlphabetic ⇒ Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), or Nl (letter number) in the Unicode Character Database

ContinuingIdentifierCharacterOrEscape ⇒
   ContinuingIdentifierCharacter
| \ *HexEscape*

ContinuingIdentifierCharacter ⇒ *UnicodeAlphanumeric* | $ | _

UnicodeAlphanumeric ⇒ Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), Nd (decimal number), Nl (letter number), Mn (non-spacing mark), Mc (combining spacing mark), or Pc (connector punctuation) in the Unicode Character Database

Semantics

LexChar[*InitialIdentifierCharacterOrEscape* ⇒ *InitialIdentifierCharacter*]
Return the character *InitialIdentifierCharacter*.

LexChar[*InitialIdentifierCharacterOrEscape* ⇒ \ *HexEscape*]
Let *ch* be the character *LexChar*[*HexEscape*].
If *ch* is in the set of characters accepted by the nonterminal *InitialIdentifierCharacter*, then return *ch*.
Signal a syntax error.

LexChar[*ContinuingIdentifierCharacterOrEscape* ⇒ *ContinuingIdentifierCharacter*]
Return the character *ContinuingIdentifierCharacter*.

LexChar[*ContinuingIdentifierCharacterOrEscape* ⇒ \ *HexEscape*]
Let *ch* be the character *LexChar*[*HexEscape*].
If *ch* is in the set of characters accepted by the nonterminal *ContinuingIdentifierCharacter*, then return *ch*.
Signal a syntax error.

The characters in the specified categories in version 2.1 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations; however, conforming ECMAScript implementations may allow additional legal identifier characters based on the category assignment from later versions of Unicode.

NOTE Identifiers are interpreted according to the grammar given in Section 5.16 of version 3.0 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode standard. This standard specifies one departure from the grammar given in the Unicode standard: $ and _ are permitted anywhere in an identifier. $ is intended for use only in mechanically generated code.

Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier. An escape sequence cannot be used to put a character into an identifier that would otherwise be illegal in that position of the identifier.

Two identifiers that are canonically equivalent according to the Unicode standard are *not* equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

7.6 Punctuators

Syntax

Punctuator ⇒
  ! | != | !== | # | % | %= | &
| && | &&= | &= | (|) | * | *=
| + | ++ | += | , | - | -- | -=
| -> | . | .. | ... | : | :: | ;
| < | << | <<= | <= | = | == | ===
| > | >= | >> | >>= | >>> | >>>= | ?
| @ | [|] | ^ | ^= | ^^ | ^^=
| { | | | |= | || | ||= | } | ~

DivisionPunctuator ⇒
  / [lookahead∉ {/, *}]
| /=

Semantics

Lex[*Punctuator*]
Return a **punctuator** token with string contents *Punctuator*.

Lex[*DivisionPunctuator*]
Return a **punctuator** token with string contents *DivisionPunctuator*.

7.7 Numeric literals

Syntax

NumericLiteral ⇒
  DecimalLiteral
| *HexIntegerLiteral* [lookahead∉ {*HexDigit*}]

DecimalLiteral ⇒
  Mantissa
| *Mantissa LetterE SignedInteger*

LetterE ⇒ E | e

Mantissa ⇒
  DecimalIntegerLiteral
| *DecimalIntegerLiteral* .
| *DecimalIntegerLiteral* . *DecimalDigits*
| . *Fraction*

DecimalIntegerLiteral ⇒
  0
| *NonZeroDecimalDigits*

NonZeroDecimalDigits ⇒
  NonZeroDigit
| *NonZeroDecimalDigits ASCIIDigit*

SignedInteger ⇒
  DecimalDigits
| + *DecimalDigits*
| - *DecimalDigits*

DecimalDigits ⇒
  ASCIIDigit
| *DecimalDigits ASCIIDigit*

HexIntegerLiteral ⇒
  `0` *LetterX HexDigit*
| *HexIntegerLiteral HexDigit*

LetterX ⇒ `X` | `x`

ASCIIDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

NonZeroDigit ⇒ `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

HexDigit ⇒ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

Semantics

Lex[*NumericLiteral* ⇒ *DecimalLiteral*]
  Return a **number** token with numeric contents *LexNumber*[*DecimalLiteral*].

Lex[*NumericLiteral* ⇒ *HexIntegerLiteral* [lookahead∉ {*HexDigit*}]]
  Return a **number** token with numeric contents *LexNumber*[*HexIntegerLiteral*].

NOTE Note that all digits of hexadecimal literals are significant.

LexNumber[*DecimalLiteral* ⇒ *Mantissa*] = *LexNumber*[*Mantissa*]

LexNumber[*DecimalLiteral* ⇒ *Mantissa LetterE SignedInteger*]
  Let *e* = *LexNumber*[*SignedInteger*].
  Return *LexNumber*[*Mantissa*]*10^{e}.

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral*] = *LexNumber*[*DecimalIntegerLiteral*]

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral* `.`] = *LexNumber*[*DecimalIntegerLiteral*]

LexNumber[*Mantissa* ⇒ *DecimalIntegerLiteral* `.` *Fraction*]
  Return *LexNumber*[*DecimalIntegerLiteral*] + *LexNumber*[*Fraction*].

LexNumber[*Mantissa* ⇒ `.` *Fraction*] = *LexNumber*[*Fraction*]

LexNumber[*DecimalIntegerLiteral* ⇒ `0`] = 0

LexNumber[*DecimalIntegerLiteral* ⇒ *NonZeroDecimalDigits*] = *LexNumber*[*NonZeroDecimalDigits*]

LexNumber[*NonZeroDecimalDigits* ⇒ *NonZeroDigit*] = *LexNumber*[*NonZeroDigit*]

LexNumber[*NonZeroDecimalDigits* ⇒ $NonZeroDecimalDigits_1$ *ASCIIDigit*]
  = 10**LexNumber*[$NonZeroDecimalDigits_1$] + *LexNumber*[*ASCIIDigit*]

LexNumber[*Fraction* ⇒ *DecimalDigits*]
  Let *n* be the number of characters in *DecimalDigits*.
  Return *LexNumber*[*DecimalDigits*]/10^{n}.

LexNumber[*SignedInteger* ⇒ *DecimalDigits*] = *LexNumber*[*DecimalDigits*]

LexNumber[*SignedInteger* ⇒ `+` *DecimalDigits*] = *LexNumber*[*DecimalDigits*]

LexNumber[*SignedInteger* ⇒ `-` *DecimalDigits*] = –*LexNumber*[*DecimalDigits*]

LexNumber[*DecimalDigits* ⇒ *ASCIIDigit*] = *LexNumber*[*ASCIIDigit*]

LexNumber[*DecimalDigits* ⇒ $DecimalDigits_1$ *ASCIIDigit*]
 = 10***LexNumber*[$DecimalDigits_1$] + *LexNumber*[*ASCIIDigit*]

LexNumber[*HexIntegerLiteral* ⇒ `0` *LetterX* *HexDigit*] = *LexNumber*[*HexDigit*]

LexNumber[*HexIntegerLiteral* ⇒ $HexIntegerLiteral_1$ *HexDigit*]
 = 16***LexNumber*[$HexIntegerLiteral_1$] + *LexNumber*[*HexDigit*]

LexNumber[*ASCIIDigit*]
 Return *ASCIIDigit*’s decimal value (a number between 0 and 9).

LexNumber[*NonZeroDigit*]
 Return *NonZeroDigit*’s decimal value (a number between 1 and 9).

LexNumber[*HexDigit*]
 Return *HexDigit*’s value (a number between 0 and 15). The letters `A`, `B`, `C`, `D`, `E`, and `F`, in either upper or lower case, have values 10, 11, 12, 13, 14, and 15, respectively.

7.8 String literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence starting with a backslash.

Syntax

The grammar parameter θ can be either single or double.

StringLiteral ⇒
 `'` $StringChars^{single}$ `'`
 | `"` $StringChars^{double}$ `"`

$StringChars^{\theta}$ ⇒
 «empty»
 | $StringChars^{\theta}$ $StringChar^{\theta}$
 | $StringChars^{\theta}$ *NullEscape* (*NullEscape*: 7.5)

$StringChar^{\theta}$ ⇒
 $LiteralStringChar^{\theta}$
 | `\` *StringEscape*

$LiteralStringChar^{single}$ ⇒ *NonTerminator* **except** `'` | `\` (*NonTerminator*: 7.4)

$LiteralStringChar^{double}$ ⇒ *NonTerminator* **except** `"` | `\`

StringEscape ⇒
 ControlEscape
 | *ZeroEscape*
 | *HexEscape*
 | *IdentityEscape*

IdentityEscape ⇒ *NonTerminator* **except** `_` | *UnicodeAlphanumeric* (*UnicodeAlphanumeric*: 7.5)

ControlEscape ⇒ `b` | `f` | `n` | `r` | `t` | `v`

ZeroEscape ⇒ `0` [lookahead∉ {*ASCIIDigit*}] (*ASCIIDigit*: 7.7)

HexEscape ⇒
  `x` *HexDigit HexDigit* (*HexDigit*: 7.7)
| `u` *HexDigit HexDigit HexDigit HexDigit*

Semantics

Lex[*StringLiteral* ⇒ `'` $StringChars^{single}$ `'`]
  Return a **string** token with string contents *LexString*[$StringChars^{single}$].

Lex[*StringLiteral* ⇒ `"` $StringChars^{double}$ `"`]
  Return a **string** token with string contents *LexString*[$StringChars^{double}$].

LexString[$StringChars^{\theta}$ ⇒ «empty»] = “”

LexString[$StringChars^{\theta}$ ⇒ $StringChars^{\theta}_{1}$ $StringChar^{\theta}$]
  Return a string consisting of the string *LexString*[$StringChars^{\theta}_{1}$] concatenated with the character *LexChar*[$StringChar^{\theta}$].

LexString[$StringChars^{\theta}$ ⇒ $StringChars^{\theta}_{1}$ *NullEscape*] = *LexString*[$StringChars^{\theta}_{1}$]

LexChar[$StringChar^{\theta}$ ⇒ $LiteralStringChar^{\theta}$]
  Return the character $LiteralStringChar^{\theta}$.

LexChar[$StringChar^{\theta}$ ⇒ `\` *StringEscape*] = *LexChar*[*StringEscape*]

LexChar[*StringEscape* ⇒ *ControlEscape*] = *LexChar*[*ControlEscape*]

LexChar[*StringEscape* ⇒ *ZeroEscape*] = *LexChar*[*ZeroEscape*]

LexChar[*StringEscape* ⇒ *HexEscape*] = *LexChar*[*HexEscape*]

LexChar[*StringEscape* ⇒ *IdentityEscape*]
  Return the character *IdentityEscape*.

NOTE A backslash followed by a non-alphanumeric character *c* other than _ or a line break represents character *c*.

LexChar[*ControlEscape* ⇒ `b`] = ‘«BS»’

LexChar[*ControlEscape* ⇒ `f`] = ‘«FF»’

LexChar[*ControlEscape* ⇒ `n`] = ‘«LF»’

LexChar[*ControlEscape* ⇒ `r`] = ‘«CR»’

LexChar[*ControlEscape* ⇒ `t`] = ‘«TAB»’

LexChar[*ControlEscape* ⇒ `v`] = ‘«VT»’

LexChar[*ZeroEscape* ⇒ `0` [lookahead∉ {*ASCIIDigit*}]] = ‘«NUL»’

LexChar[*HexEscape* ⇒ `x` $HexDigit_1$ $HexDigit_2$]
  Let *n* = 16**LexNumber*[$HexDigit_1$] + *LexNumber*[$HexDigit_2$].
  Return the character with code point value *n*.

LexChar[*HexEscape* ⇒ `u` $HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$]

Let *n* = 4096**LexNumber*[$HexDigit_1$] + 256**LexNumber*[$HexDigit_2$] + 16**LexNumber*[$HexDigit_3$] + *LexNumber*[$HexDigit_4$].

Return the character with code point value *n*.

NOTE A *LineTerminator* character cannot appear in a string literal, even if preceded by a backslash `\`. The correct way to cause a line terminator character to be part of the string value of a string literal is to use an escape sequence such as `\n` or `\u000A`.

7.9 Regular expression literals

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegExpBody* and the *RegExpFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it should not extend the *RegExpBody* and *RegExpFlags* productions or the productions used by these productions.

Syntax

RegExpLiteral ⇒ *RegExpBody* *RegExpFlags*

RegExpFlags ⇒
  «empty» (*ContinuingIdentifierCharacterOrEscape*: 7.5)
| *RegExpFlags* *ContinuingIdentifierCharacterOrEscape*
| *RegExpFlags* *NullEscape* (*NullEscape*: 7.5)

RegExpBody ⇒ `/` [lookahead∉ {`*`}] *RegExpChars* `/`

RegExpChars ⇒
  RegExpChar
| *RegExpChars* *RegExpChar*

RegExpChar ⇒
  OrdinaryRegExpChar
| `\` *NonTerminator* (*NonTerminator*: 7.4)

OrdinaryRegExpChar ⇒ *NonTerminator* **except** `\` | `/`

Semantics

Lex[*RegExpLiteral* ⇒ *RegExpBody* *RegExpFlags*]

Return a **regularExpression** token with the body string *LexString*[*RegExpBody*] and flags string *LexString*[*RegExpFlags*].

LexString[*RegExpFlags* ⇒ «empty»] = “”

LexString[*RegExpFlags* ⇒ $RegExpFlags_1$ *ContinuingIdentifierCharacterOrEscape*]

Return a string consisting of the string *LexString*[$RegExpFlags_1$] concatenated with the character *LexChar*[*ContinuingIdentifierCharacterOrEscape*].

LexString[*RegExpFlags* ⇒ $RegExpFlags_1$ *NullEscape*] = *LexString*[$RegExpFlags_1$]

LexString[*RegExpBody* ⇒ `/` [lookahead∉ {`*`}] *RegExpChars* `/`] = *LexString*[*RegExpChars*]

LexString[*RegExpChars* ⇒ *RegExpChar*] = *LexString*[*RegExpChar*]

LexString[*RegExpChars* ⇒ *RegExpChars*$_1$ *RegExpChar*]

Return a string consisting of the string *LexString*[*RegExpChars*$_1$] concatenated with the string *LexString*[*RegExpChar*].

LexString[*RegExpChar* ⇒ *OrdinaryRegExpChar*]

Return a string consisting of the single character *OrdinaryRegExpChar*.

LexString[*RegExpChar* ⇒ `\` *NonTerminator*]

Return a string consisting of the two characters '`\`' and *NonTerminator*.

NOTE A regular expression literal is an input element that is converted to a RegExp object (section *****) when it is scanned. The object is created before evaluation of the containing program or function begins. Evaluation of the literal produces a reference to that object; it does not create a new object. Two regular expression literals in a program evaluate to regular expression objects that never compare as **`===`** to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by **`new RegExp`** (section *****) or calling the **`RegExp`** constructor as a function (section *****).

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters `//` start a single-line comment. To specify an empty regular expression, use `/(?:)/`.

8 Program Structure

8.1 Packages

8.2 Scopes

9 Data Model

This chapter describes the essential state held in various ECMAScript objects. This state is presented abstractly using the formalisms from chapter 5. Much of the state held in these objects is observable by ECMAScript programmers only indirectly, and implementation are encouraged to implement these objects in other ways as long as the observable behaviour is the same as described here.

9.1 Objects

An object is a first-class data value visible to ECMAScript programmers. Every object is either **undefined**, **null**, a boolean, a number, a string, a namespace, an attribute, a class, a method closure, or a general instance. These kinds of objects are described in the subsections below.

OBJECT is the set of all possible objects and is defined as:

OBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE

9.1.1 Undefined

There is exactly one **undefined** value. The set UNDEFINED consists of that one value.

UNDEFINED = {**undefined**}

9.1.2 Null

There is exactly one **null** value. The set NULL consists of that one value.

NULL = {**null**}

9.1.3 Booleans

There are two booleans, **true** and **false**. The set BOOLEAN consists of these two values. See section 5.3.

9.1.4 Numbers

The set FLOAT64 consists of all representable double-precision floating-point IEEE 754 values. See section 5.6.

9.1.5 Strings

The set STRING consists of all representable strings. See section 5.9. A STRING *s* is considered to be of either the class `String` if *s*'s length isn't 1 or the class `Character` if *s*'s length is 1.

9.1.6 Namespaces

A namespace object is represented by a record (see section 5.11) with tag **namespace** and the field below. Each time a namespace is created, the new namespace is different from every other namespace, even if it happens to share the name of an existing namespace. NAMESPACE is the set of all possible **namespace** records.

Field	Contents	Note
name	STRING	The namespace's name used by toString

NAMESPACEOPT consists of all namespaces as well as **null**:

NAMESPACEOPT = NULL ∪ NAMESPACE

9.1.7 Attributes

Attribute objects are values obtained from combining zero or more syntactic attributes (see *****). An attribute object is represented by a tuple (see section 5.10) with tag **attribute** and the fields below. ATTRIBUTE is the set of all possible **attribute** tuples.

Field	Contents	Note
namespaces	NAMESPACE{}	The set of namespaces contained in this attribute
local	BOOLEAN	**true** if the `local` attribute has been given
extend	CLASSOPT	A class if the `extend` attribute has been given; **null** if not
enumerable	BOOLEAN	**true** if the `enumerable` attribute has been given
classMod	CLASSMODIFIER	**dynamic** or **fixed** if one of these attributes has been given; **null** if not. CLASSMODIFIER = {**null**, **dynamic**, **fixed**}
memberMod	MEMBERMODIFIER	**static**, **constructor**, **operator**, **abstract**, **virtual**, or **final** if one of these attributes has been given; **null** if not. MEMBERMODIFIER = {**null**, **static**, **constructor**, **operator**, **abstract**, **virtual**, **final**}
overrideMod	OVERRIDEMODIFIER	**mayOverride** or **override** if one of these attributes has been given; **null** if not. OVERRIDEMODIFIER = {**null**, **mayOverride**, **override**}
prototype	BOOLEAN	**true** if the `prototype` attribute has been given
unused	BOOLEAN	**true** if the `unused` attribute has been given

NOTE An implementation that supports host-defined attributes will add other fields to the tuple above.

9.1.8 Classes

Programmer-visible class objects are represented as records (see section 5.11) with tag **class** and the fields below. CLASS is the set of all possible **class** records.

Field	Contents	Note
super	CLASSOPT	This class's immediate superclass or **null** if none
prototype	OBJECT	An object that serves as this class's prototype for compatibility with ECMAScript 3; may be **null**
globalMembers	GLOBALMEMBER{}	A set of global members defined in this class
instanceMembers	INSTANCEMEMBER{}	A set of instance members defined in this class
definitionNamespaces	NAMESPACE{}	The set of namespaces in the attributes prefixing this class's definition
classMod	CLASSMODIFIER	**dynamic** if this class allows dynamic properties; **null** if this class doesn't allow dynamic properties but its proper descendants may; **fixed** if neither this class nor its descendants can allow dynamic properties
primitive	BOOLEAN	**true** if this class was defined with the `primitive` attribute
privateNamespace	NAMESPACE	This class's `private` namespace
call	INVOKER	A function to call when this class is used in a call expression
construct	INVOKER	A function to call when this class is used in a `new` expression

CLASSOPT consists of all classes as well as **null**:

CLASSOPT = NULL ∪ CLASS

INVOKER is the set of functions that take an OBJECT (the `this` value), a list of OBJECTs (the positional arguments), and a set of NAMEDARGUMENTs (the named arguments) and produce an OBJECT result.

INVOKER = OBJECT × OBJECT[] × NAMEDARGUMENT{} → OBJECT

A CLASS c is an *ancestor* of CLASS d if either $c = d$ or d.**super** $= s$, $s \neq$ **null**, and c is an ancestor of s. A CLASS c is a *descendant* of CLASS d if d is an ancestor of c.

A CLASS c is a *proper ancestor* of CLASS d if both c is an ancestor of d and $c \neq d$. A CLASS c is a *proper descendant* of CLASS d if d is a proper ancestor of c.

9.1.8.1 Members

A record (see section 5.11) with tag **globalMember** and the fields below controls the behaviour of either reading or writing a property of an instance of a class. GLOBALMEMBER is the set of all possible **globalMember** tuples.

Field	Contents	Note
name	STRING	The member's unqualified name
namespaces	NAMESPACE{}	The set of namespaces qualifying **name**. This set is never empty.
category	GLOBALCATEGORY	The member's category. GLOBALCATEGORY = {**static**,

		constructor}
readable	BOOLEAN	**true** if this member is visible in read accesses
writable	BOOLEAN	**true** if this member is visible in write accesses
indexable	BOOLEAN	**true** if this member can be accessed via the `[]` indexing operator
enumerable	BOOLEAN	**true** if this member is visible in a `for-in` loop
data	GLOBALDATA ∪ NAMESPACE	Information about how to get or set this member's value. GLOBALDATA = GLOBALSLOT ∪ METHOD ∪ ACCESSOR. A GLOBALSLOT is the slot holding this member; a METHOD specifies that this member is a method; an ACCESSOR contains code to run to access this member; and a NAMESPACE *n* indicates that this member is an alias of another member with the same unqualified name and namespace *n*.

A record (see section 5.11) with tag **instanceMember** and the fields below controls the behaviour of either reading or writing a property of an instance of a class. INSTANCEMEMBER is the set of all possible **instanceMember** tuples.

Field	Contents	Note
name	STRING	The member's unqualified name
namespaces	NAMESPACE{}	The set of namespaces qualifying **name**. This set is never empty.
category	INSTANCECATEGORY	The member's category. INSTANCECATEGORY = {**abstract**, **virtual**, **final**}
readable	BOOLEAN	**true** if this member is visible in read accesses
writable	BOOLEAN	**true** if this member is visible in write accesses
indexable	BOOLEAN	**true** if this member can be accessed via the `[]` indexing operator
enumerable	BOOLEAN	**true** if this member is visible in a `for-in` loop
data	INSTANCEDATA ∪ NAMESPACE	Information about how to get or set this member's value. INSTANCEDATA = SLOTID ∪ METHOD ∪ ACCESSOR. A SLOTID names the instance slot holding this member; a METHOD specifies that this member is a method; an ACCESSOR contains code to run to access this member; and a NAMESPACE *n* indicates that this member is an alias of another member with the same unqualified name and namespace *n*.

The following sets are unions of their instance and global equivalents:

MEMBER = INSTANCEMEMBER ∪ GLOBALMEMBER

MEMBERDATA = INSTANCEDATA ∪ GLOBALDATA

MEMBERDATAOPT = NULL ∪ MEMBERDATA

A record (see section 5.11) with tag **method** and the fields below describes a non-accessor member defined with the `function` keyword. METHOD is the set of all possible **method** tuples.

Field	Contents	Note
type	SIGNATURE	The method's signature (see 9.3)

f	INSTANCEOPT	A callable object or **null** if this is an abstract method

A record (see section 5.11) with tag **accessor** and the fields below describes an accessor — a member defined with the `function get` or `function set` keywords that runs code to do the read or write. ACCESSOR is the set of all possible **accessor** tuples.

Field	Contents	Note
type	CLASS	The type of the value that can be read or written by this member
f	INSTANCE	A callable object; calling this object does the read or write

9.1.9 Method Closures

A tuple (see section 5.10) with tag **methodClosure** and the fields below describes an instance method with a bound `this` value. METHODCLOSURE is the set of all possible **methodClosure** tuples.

Field	Contents	Note
this	OBJECT	The bound `this` value
method	METHOD	The bound method

9.1.10 General Instances

Instances of programmer-defined classes as well as of some built-in classes are represented as records (see section 5.11) with tag **instance** and the fields below. INSTANCE is the set of all possible **instance** records.

NOTE Instances of some built-in classes are represented as described in sections 9.1.1 through 9.1.9 rather than as **instance** records. This distinction is made for convenience in specifying the language's behaviour and is invisible to the programmer.

Field	Contents	Note
type	CLASS	This instance's type
model	INSTANCEOPT	If this instance was created by calling `new` on a `prototype` function, the value of the function's `prototype` property at the time of the call; **null** otherwise.
call	INVOKER	A function to call when this instance is used in a call expression
construct	INVOKER	A function to call when this instance is used in a `new` expression
typeofString	STRING	A string to return if `typeof` is invoked on this instance
slots	SLOT{}	A set of slots that hold this instance's fixed property values
dynamicProperties	DYNAMICPROPERTY{}	A set of this instance's dynamic properties

INSTANCEOPT consists of all **instance** records as well as **null**:

INSTANCEOPT = NULL ∪ INSTANCE

A record (see section 5.11) with tag **dynamicProperty** and the fields below describes one dynamic property of one instance. DYNAMICPROPERTY is the set of all possible **dynamicProperty** records.

Field	Contents	Note

name	STRING	This dynamic property's name
value	OBJECT	This dynamic property's current value

9.1.10.1 Slots

A record (see section 5.11) with tag **slot** and the fields below describes the value of one fixed property of one instance. SLOT is the set of all possible **slot** records.

Field	Contents	Note
id	SLOTID	A unique identifier used to look up this slot
value	OBJECT	This fixed property's current value

A record (see section 5.11) with tag **slotId** and the field below serves as a unique identifier that distinguishes one member's slots from another member's. SLOTID is the set of all possible **slotId** records.

Field	Contents	Note
type	CLASS	The type of values that can be stored in this slot

9.2 Qualified Names

A tuple (see section 5.10) with tag **qualifiedName** and the fields below represents a fully qualified name. QUALIFIEDNAME is the set of all possible **qualifiedName** tuples.

Field	Contents	Note
namespace	NAMESPACE	The namespace qualifier
name	STRING	The name

9.3 Signatures

A tuple (see section 5.10) with tag **signature** and the fields below represents the type signature of a function. SIGNATURE is the set of all possible **signature** tuples.

Field	Contents	Note
requiredPositional	CLASS[]	List of the types of the required positional parameters
optionalPositional	CLASS[]	List of the types of the optional positional parameters, which follow the required positional parameters
requiredNamed	NAMEDPARAMETER{}	Set of the types and names of the required named parameters, which follow the positional parameters
optionalNamed	NAMEDPARAMETER{}	Set of the types and names of the optional named parameters
rest	CLASSOPT	The type of any extra arguments that may be passed or **null** if no extra arguments are allowed
restAllowsNames	BOOLEAN	**true** if the extra arguments may be named
returnType	CLASS	The type of this function's result

A tuple (see section 5.10) with tag **namedParameter** and the fields below represents the signature of one named parameter. NAMEDPARAMETER is the set of all possible **namedParameter** tuples.

Field	Contents	Note
name	STRING	This parameter's name

type	CLASS	This parameter's type

9.4 Unary Operators

There are ten global tables for dispatching unary operators. These tables are the *plusTable*, *minusTable*, *bitwiseNotTable*, *incrementTable*, *decrementTable*, *callTable*, *constructTable*, *bracketReadTable*, *bracketWriteTable*, and *bracketDeleteTable*. Each of these tables is an independent record (see section 5.11) with tag **unaryTable** and the field below. UNARYTABLE is the set of all **unaryTable** records.

Field	Contents	Note
methods	UNARYMETHOD{}	A set of defined unary methods

A tuple (see section 5.10) with tag **unaryMethod** and the fields below represents unary operator method. UNARYMETHOD is the set of all possible **unaryMethod** tuples.

Field	Contents	Note
operandType	CLASS	The dispatched operand's type
op	OBJECT × OBJECT × OBJECT[] × NAMEDARGUMENT[] → OBJECT	Function that takes a `this` value, a first positional argument, a list of other positional arguments, and a set of named arguments and returns the operator's result

9.5 Binary Operators

There are fifteen global tables for dispatching binary operators. These tables are the *addTable*, *subtractTable*, *multiplyTable*, *divideTable*, *remainderTable*, *lessTable*, *lessOrEqualTable*, *equalTable*, *strictEqualTable*, *shiftLeftTable*, *shiftRightTable*, *shiftRightUnsignedTable*, *bitwiseAndTable*, *bitwiseXorTable*, and *bitwiseOrTable*. Each of these tables is an independent record (see section 5.11) with tag **binaryTable** and the field below. BINARYTABLE is the set of all **binaryTable** records.

Field	Contents	Note
methods	BINARYMETHOD{}	A set of defined binary methods

A tuple (see section 5.10) with tag **binaryMethod** and the fields below represents binary operator method. BINARYMETHOD is the set of all possible **binaryMethod** tuples.

Field	Contents	Note
leftType	CLASS	The left operand's type
rightType	CLASS	The right operand's type
op	OBJECT × OBJECT → OBJECT	Function that takes the left and right operand values and returns the operator's result

10 Data Operations

This chapter describes core algorithms defined on the values in chapter 9. The algorithms here are not ECMAScript language construct themselves; rather, they are called as subroutines in computing the effects of the language constructs presented in later chapters. The algorithms are optimised for ease of presentation and understanding rather than speed, and implementations are encouraged to implement these algorithms more efficiently as long as the observable behaviour is as described here.

10.1 Name Lookup

10.2 Member Lookup

10.2.1 Reading a Qualified Property

readQualifiedProperty(*o*, *name*, *ns*, *indexableOnly*) reads the property *ns*`::`*name* of object *o* and returns the value of the property. If *indexableOnly* is true, only `indexable` properties are considered.

```
function readQualifiedProperty(o: OBJECT, name: STRING, ns: NAMESPACE,
        indexableOnly: BOOLEAN): OBJECT
   if o ∈ INSTANCE then
      if ns = publicNamespace and
            there exists a p ∈ o.dynamicProperties such that name = p.name then
         return p.value
      end if;
      if o.model ≠ null then
         return readQualifiedProperty(o.model, name, ns, indexableOnly)
      end if
   end if;
   d: MEMBERDATAOPT ← null;
   if o ∈ CLASS then d ← mostSpecificMember(o, true, name, ns, indexableOnly)
   else d ← mostSpecificMember(objectType(o), false, name, ns, indexableOnly)
   end if;
   case d of
      {null} do
         if objectType(o).classMod = dynamic then return undefined end if;
         throw propertyNotFoundError;
      GLOBALSLOT do return d.value;
      SLOTID do
         At this point o is guaranteed to be an instance that has a unique slot s such that s.id = d.
         return s.value;
      METHOD do return methodClosure⟨o, d⟩;
      ACCESSOR do return d.f.call(o, [], {})
   end case
```

mostSpecificMember(*c*, *global*, *name*, *ns*, *indexableOnly*) searches for a global (if global is true) or instance (if global is false) member *ns*`::`*name* in class *c* and its ancestors. If *indexableOnly* is true, only `indexable` members are considered. If class *c* and its ancestors contain several definitions of *ns*`::`*name*, the one in the most derived class is chosen. If found, *mostSpecificMember* returns a MEMBERDATA record; if not found, *mostSpecificMember* returns **null**.

function *mostSpecificMember*(*c*: CLASS, *global*: BOOLEAN, *name*: STRING, *ns*: NAMESPACE,
  indexableOnly: BOOLEAN): MEMBERDATAOPT
  ns2: NAMESPACE ← *ns*;
  members: MEMBER{} ← *c*.**instanceMembers**;
  if *global* **then** *members* ← *c*.**globalMembers** **end if**;
  if there exists a *m* ∈ *members* such that:
      m.**readable** is **true**,
      name = *m*.**name**,
      ns ∈ *m*.**namespaces**, and
      either *indexableOnly* is **false** or *m*.**indexable** is **true then**
    d: MEMBERDATA ∪ NAMESPACE ← *m*.**data**;
    if *d* ∉ NAMESPACE **then return** *d* **end if**;
    ns2 ← *d*
  end if;
  s: CLASSOPT ← *c*.**super**;
  if *s* ≠ **null then return** *mostSpecificMember*(*s*, *global*, *name*, *ns2*, *indexableOnly*) **end if**;
  return null

10.2.2 Reading an Unqualified Property

readUnqualifiedProperty(*o*, *name*, *uses*) reads the unqualified property *name* of object *o* and returns the value of the property. *uses* is a set of namespaces `use`d around the point of the reference.

readUnqualifiedProperty works by calling *resolveObjectNamespace* to find a namespace and then proceeds as in reading a qualified property.

function *readUnqualifiedProperty*(*o*: OBJECT, *name*: STRING, *uses*: NAMESPACE{}): OBJECT
  ns: NAMESPACE ← *resolveObjectNamespace*(*o*, *name*, *uses*);
  return *readQualifiedProperty*(*o*, *name*, *ns*, **false**)

resolveObjectNamespace(*o*, *name*, *uses*) finds a namespace to use when reading an unqualified property by searching for a member in the *least* derived ancestor that matches the name and has one of the namespaces in the *uses* set. If no member is found, *resolveObjectNamespace* returns the `public` namespace.

function *resolveObjectNamespace*(*o*: OBJECT, *name*: STRING, *uses*: NAMESPACE{}): NAMESPACE
  if *o* ∈ INSTANCE **and** *o*.**model** ≠ **null then**
    return *resolveObjectNamespace*(*o*.**model**, *name*, *uses*)
  end if;
  ns: NAMESPACEOPT ← **null**;
  if *o* ∈ CLASS **then** *ns* ← *resolveMemberNamespace*(*o*, **true**, *name*, *uses*)
  else *ns* ← *resolveMemberNamespace*(*objectType*(*o*), **false**, *name*, *uses*)
  end if;
  if *ns* ≠ **null then return** *ns* **end if**;
  return *publicNamespace*

mostSpecificMember(*c*, *global*, *name*, *ns*, *indexableOnly*) searches for a global (if global is true) or instance (if global is false) member *ns*`::`*name* in class *c* and its ancestors. If *indexableOnly* is true, only `indexable` members are considered. If class *c* and its ancestors contain several definitions of *ns*`::`*name*, the one in the most derived class is chosen. If found, *mostSpecificMember* returns a MEMBERDATA record; if not found, *mostSpecificMember* returns **null**.

function *resolveMemberNamespace*(*c*: CLASS, *global*: BOOLEAN, *name*: STRING, *uses*: NAMESPACE{}):
NAMESPACEOPT
s: CLASSOPT ← *c*.**super**;
if *s* ≠ **null then**
ns: NAMESPACEOPT ← *resolveMemberNamespace*(*s*, *global*, *name*, *uses*);
if *ns* ≠ **null then return** *ns* **end if**
end if;
members: MEMBER{} ← *c*.**instanceMembers**;
if *global* **then** *members* ← *c*.**globalMembers end if**;
Let *matches*: MEMBER{} be the set of all *m* ∈ *members* such that:
m.**readable** is **true**,
name = *m*.**name**, and
uses ∩ *m*.**namespaces** ≠ **{}**.
if *matches* ≠ **{} then**
if |*matches*| > 1 **then**
This access is ambiguous because it found several different members in the same class.
throw propertyNotFoundError
end if;
Let *match*: MEMBER be the one element of *matches*.
overlappingNamespaces: NAMESPACE{} ← *uses* ∩ *match*.**namespaces**;
Let *ns2*: NAMESPACE be any element of *overlappingNamespaces*.
return *ns2*
end if;
return null

10.3 Object Utilities

10.3.1 objectType

objectType(*o*) returns an OBJECT *o*'s most specific type.

function *objectType*(*o*: OBJECT): CLASS
case *o* **of**
UNDEFINED **do return** *undefinedClass* (see *****);
NULL **do return** *nullClass* (see *****);
BOOLEAN **do return** *booleanClass* (see *****);
FLOAT64 **do return** *numberClass* (see *****);
STRING **do**
if |*o*| = 1 **then return** *characterClass* (see *****) **end if**;
return *stringClass* (see *****);
NAMESPACE **do return** *namespaceClass* (see *****);
ATTRIBUTE **do return** *attributeClass* (see *****);
CLASS **do return** *classClass* (see *****);
METHODCLOSURE **do return** *functionClass* (see *****);
INSTANCE **do return** *o*.**type**
end case

10.3.2 instanceOf

There are two tests for determining whether an object *o* is an instance of class *c*. The first, *instanceOf*, is used for the purposes of method dispatch and helps determine whether a method of *c* can be called on *o*. The second, *relaxedInstanceOf*, determines whether *o* can be stored in a variable of type *c* without conversion.

instanceOf(*o*, *c*) returns **true** if *o* is an instance of class *c*. It considers **null** to be an instance of the classes `Null` and `Object` only.

function *instanceOf*(*o*: OBJECT, *c*: CLASS): BOOLEAN
  t: CLASS ← *objectType*(*o*);
  if *c* is an *ancestor* (see 9.1.8) of *t* **then return true**
  else return false
  end if

relaxedInstanceOf(*o*, *c*) returns **true** if *o* is an instance of class *c* but considers **null** to be an instance of the classes `Null`, `Object`, and all other non-primitive classes.

function *relaxedInstanceOf*(*o*: OBJECT, *c*: CLASS): BOOLEAN
  t: CLASS ← *objectType*(*o*);
  if *o* = **null and not** *c*.**primitive then return true end if**;
  return *instanceOf*(*o*, *c*)

10.3.3 toBoolean

toBoolean(*o*) coerces an object *o* to a boolean.

function *toBoolean*(*o*: OBJECT): BOOLEAN
  case *o* **of**
    UNDEFINED ∪ NULL **do return false**;
    BOOLEAN **do return** *o*;
    FLOAT64 **do return** *o* ∉ {**+zero**, **–zero**, **NaN**};
    STRING **do return** *o* ≠ “”;
    NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE **do return true**;
    INSTANCE **do** *****
  end case

10.3.4 toNumber

toNumber(*o*) coerces an object *o* to a number.

function *toNumber*(*o*: OBJECT): FLOAT64
  case *o* **of**
    UNDEFINED **do return NaN**;
    NULL ∪ {**false**} **do return +zero**;
    {**true**} **do return** 1.0;
    FLOAT64 **do return** *o*;
    STRING **do** *****;
    NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE **do throw typeError**;
    UNDEFINED ∪ NULL **do return false**;
    INSTANCE **do** *****
  end case

10.3.5 toString

toString(*o*) coerces an object *o* to a string.

function *toString*(*o*: OBJECT): STRING
  case *o* **of**
    UNDEFINED **do return** "`undefined`";
    NULL **do return** "`null`";
    {**false**} **do return** "`false`";
    {**true**} **do return** "`true`";
    FLOAT64 **do** *****;
    STRING **do return** *o*;
    NAMESPACE **do** *****;
    ATTRIBUTE **do** *****;
    CLASS **do** *****;
    METHODCLOSURE **do** *****;
    INSTANCE **do** *****
  end case

10.4 Unary Operator Dispatch

unaryDispatch(*table*, *limit*, *this*, *op*, *positionalArgs*, *namedArgs*) dispatches the unary operator described by *table* applied to the `this` value *this*, the first argument *op*, a vector of zero or more additional positional arguments *positionalArgs*, and a set of zero or more named arguments *namedArgs*. If *limit* is non-**null**, lookup is restricted to operators defined on the proper superclasses of *limit*.

function *unaryDispatch*(*table*: UNARYTABLE, *limit*: CLASSOPT, *this*: OBJECT, *op*: OBJECT,
    positionalArgs: OBJECT[], *namedArgs*: NAMEDARGUMENT{}): OBJECT
  Let *applicableMethods*: UNARYMETHOD{} be the set of all *m* ∈ *table*.**methods** such that
      limitedInstanceOf(*op*, *m*.**operandType**, *limit*) = **true**.
  Let *bestMethods*: UNARYMETHOD{} be the set of all *m* ∈ *applicableMethods* such that
      given the choice of *m*, for every *m2* ∈ *applicableMethods*, *m2* is an *ancestor* (see 9.1.8) of *m*.
  if |*bestMethods*| = 0 **then throw methodNotFoundError end if**
  At this point *bestMethods* must contain exactly one element. Let *b*: UNARYMETHOD be that element.
  return *b*.**op**(*this*, *op*, *positionalArgs*, *namedArgs*)

limitedInstanceOf(*v*, *c*, *limit*) returns **true** if *v* is a member of class *c* with the added condition that, if *limit* is non-**null**, *c* is a proper superclass of *limit*.

function *limitedInstanceOf*(*v*: OBJECT, *c*: CLASS, *limit*: CLASSOPT): BOOLEAN
  if *instanceOf*(*v*, *c*) **then**
    if *limit* = **null** **or** *c* is a *proper ancestor* (see 9.1.8) of *limit* **then return true**
    else return false
    end if
  else return false
  end if

10.5 Binary Operator Dispatch

m1: BINARYMETHOD is *at least as specific as* *m2*: BINARYMETHOD if *m2*.**leftType** is an *ancestor* (see 9.1.8) of *m1*.**leftType** and *m2*.**rightType** is an *ancestor* of *m1*.**rightType**.

binaryDispatch(*table*, *leftLimit*, *rightLimit*, *left*, *right*) dispatches the binary operator specified by *table* applied to the operands *left* and *right*. If *leftLimit* is non-**null**, the lookup is restricted to operator definitions with a superclass of *leftLimit* for the left operand. Similarly, if *rightLimit* is non-**null**, the lookup is restricted to operator definitions with a superclass of *rightLimit* for the right operand.

function *binaryDispatch*(*table*: BINARYTABLE, *leftLimit*: CLASSOPT, *rightLimit*: CLASSOPT,
    left: OBJECT, *right*: OBJECT): OBJECT
  Let *applicableMethods*: BINARYMETHOD{} be the set of all *m* ∈ *table*.**methods** such that
      limitedInstanceOf(*left*, *m*.**leftType**, *leftLimit*) = **true** and
      limitedInstanceOf(*right*, *m*.**rightType**, *rightLimit*) = **true**.
  Let *bestMethods*: BINARYMETHOD{} be the set of all *m* ∈ *applicableMethods* such that
      given the choice of *m*, for every *m2* ∈ *applicableMethods*, *m* is *at least as specific as* *m2*.
  if |*bestMethods*| = 0 **then throw methodNotFoundError end if**
  At this point *bestMethods* must contain exactly one element. Let *b*: BINARYMETHOD be that element.
  return *b*.**op**(*left*, *right*)

11 Evaluation

11.1 Phases of Evaluation

11.2 Constant Expressions

12 Expressions

12.1 Identifiers

12.2 Qualified Identifiers

12.3 Units

12.3.1 Unit Grammar

12.4 Array Initialisers

12.5 Object Initialisers

12.6 Primary Expressions

12.7 Super Expressions

12.8 Postfix Expressions

12.9 Unary Operators

12.10 Multiplicative Operators

12.11 Additive Operators

12.12 Shift Operators

12.13 Relational Operators

12.14 Equality Operators

12.15 Bitwise Operators

12.16 Logical Operators

12.17 Conditional Operator

12.18 Assignment Operators

12.19 Comma Operator

13 Statements

13.1 Empty Statement

13.2 Expression Statement

13.3 Super Statement

13.4 Block Statement

13.5 Labelled Statement

13.6 If Statement

13.7 Switch Statement

13.8 Do-While Statement

13.9 While Statement

13.10 For Statements

13.11 With Statement

13.12 Continue Statement

13.13 Break Statement

13.14 Return Statement

13.15 Throw Statement

13.16 Try Statement

14 Directives

14.1 Annotations

14.2 Annotated Blocks

14.3 Variable Definition

14.4 Alias Definition

14.5 Function Definition

14.6 Class Definition

14.7 Namespace Definition

14.8 Package Definition

14.9 Import Directive

14.10 Namespace Use Directive

14.11 Pragmas

14.11.1 Strict Mode

15 Predefined Identifiers

16 Built-in Classes

16.1 Object

16.2 Never

16.3 Void

16.4 Null

16.5 Boolean

16.6 Integer

16.7 Number

16.7.1 ToNumber Grammar

16.8 Character

16.9 String

16.10 Function

16.11 Array

16.12 Type

16.13 Math

16.14 Date

16.15 RegExp

16.15.1 Regular Expression Grammar

16.16 Unit

16.17 Error

16.18 Attribute

17 Built-in Functions

18 Built-in Attributes

19 Built-in Operators

20 Built-in Namespaces

21 Built-in Units

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

23.3 Units