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.

The table of contents and Appendix A are new in this draft and don't have change bars.

The pervasive reformatting of code to group actions of different productions in the same rule does not have change bars.

The pervasive renames of *Constrain* to *Validate*, REFERENCE to OBJORREF, and some of the associated renames of semantic local variable names do not have change bars.

The pervasive replacement of *Super* actions by LIMITEDINSTANCE etc. types does not have change bars.

Section 5.7 has been revised extensively enough that change bars have been removed to make it readable.

Table of Contents

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 Semantic Domains

Semantic domains describe the possible values that a variable might take on in an algorithm. The algorithms are constructed in a way that ensures that these constraints are always met, regardless of any valid or invalid programmer or user input or actions.

A semantic domain can be intuitively thought of as a set of possible values, and, in fact, any set of values explicitly described in this document is also a semantic domain. Nevertheless, semantic domains have a more precise mathematical definition in domain theory (see for example David Schmidt, *Denotational Semantics: A Methodology for Language Development*; Allyn and Bacon 1986) that allows one to define semantic domains recursively without encountering paradoxes such as trying to define a set *A* whose members include all functions mapping values from *A* to INTEGER. The problem with an ordinary definition of such a set *A* is that the cardinality of the set of all functions mapping *A* to INTEGER is always strictly greater than the cardinality of *A*, leading to a contradiction. Domain theory uses a least fixed point construction to allow *A* to be defined as a semantic domain without encountering problems.

Semantic domains have names in CAPITALISED 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 entities.

A variable *v* is constrained using the notation

v: T

where T is a semantic domain. This constraint indicates that the value of *v* will always be a member of the semantic domain T. These declarations are informative (they may be dropped without affecting the semantics' correctness) but useful in understanding the semantics. For example, when the semantics state that *x*: INTEGER then one does not have to worry about what happens when *x* has the value **true** or **+∞**.

The constraints can be proven statically. The semantics have been machine-checked to ensure that every constraint holds.

5.3 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**.

5.4 Booleans

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

Let *a* and *b* be ~~booleans~~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* ≠ *b*

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

5.5 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:

{$element_1$, $element_2$, ... , $element_n$}

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

$\{f(x) \mid \forall x \in A\}$

which denotes the set of the results of computing expression f on all elements x of set A. A predicate can be added:

$\{f(x) \mid \forall x \in A$ **such that** $predicate(x)\}$

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

$\{x \mid \forall x \in$ INTEGER **such that** $x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}$
$\{x^2 \mid \forall x \in \{-5, -1, 1, 2, 4\}\} = \{1, 4, 16, 25\}$
$\{x \times 10 + y \mid \forall x \in \{1, 2, 4\}, \forall y \in \{3, 5\}\} = \{13, 15, 23, 25, 43, 45\}$

<u>The same notation is used for operations on sets and on semantic domains. Let *A* and *B* be sets (or semantic domains) and *x* and *y* be values. The following operations can be done on them:</u>~~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
$x \notin A$	**false** if x is an element of ~~set~~ A and **true** if not
$\lvert A\rvert$	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 <u>or semantic domain</u> of all values that are present both in A and in B)
$A \cup B$	The union of ~~sets~~ A and B (the set <u>or semantic domain</u> 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 <u>or semantic domain</u> 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** $A \neq B$.

<u>If T is a semantic domain, then T{} is the semantic domain of all sets whose elements are members of T. For example, if</u>

<u>T = {1,2,3}</u>

<u>then:</u>

<u>T{} = {{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}</u>

<u>The empty set {} is a member of T{} for any semantic domain T.</u>

In addition to the above, the **some** and **every** quantifiers can be used on sets. The quantifier

some $x \in A$ **satisfies** $predicate(x)$

returns **true** if there exists at least one element x in set A such that $predicate(x)$ computes to **true**. If there is no such element x, then the **some** quantifier's result is **false**. If the **some** quantifier returns **true**, then variable x is left bound to any element of A for which $predicate(x)$ computes to **true**; if there is more than one such element x, then one of them is chosen arbitrarily. For example,

some $x \in \{3, 16, 19, 26\}$ **satisfies** x **mod** $10 = 6$

evaluates to **true** and leaves x set to either 16 or 26. Other examples include:

(**some** $x \in$ {3, 16, 19, 26} **satisfies** x **mod** 10 = 7) = **false**;
(**some** $x \in$ {} **satisfies** x **mod** 10 = 7) = **false**;
(**some** $x \in$ {“`Hello`”} **satisfies** **true**) = **true** and leaves x set to the string “`Hello`”;
(**some** $x \in$ {} **satisfies** **true**) = **false**.

The quantifier

every $x \in A$ **satisfies** *predicate*(x)

returns **true** if there exists no element x in set A such that *predicate*(x) computes to **false**. If there is at least one such element x, then the **every** quantifier’s result is **false**. As a degenerate case, the **every** quantifier is always **true** if the set A is empty. For example,

(**every** $x \in$ {3, 16, 19, 26} **satisfies** x **mod** 10 = 6) = **false**;
(**every** $x \in$ {6, 26, 96, 106} **satisfies** x **mod** 10 = 6) = **true**;
(**every** $x \in$ {} **satisfies** x **mod** 10 = 6) = **true**.

5.6 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 semantic domain~~set~~ of all integers {... –3, –2, –1, 0, 1, 2, 3 ...}. 3.0, 3, 0xFF, and -10^{100} are all integers.

RATIONAL is the semantic domain~~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 semantic domain~~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 $x\neq0$ 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 $x\geq0$ 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.

5.6.1 Bitwise Integer Operators

The four procedures 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**.

More precisely, any integer x can be represented as an infinite sequence of bits a_i where the index i ranges over the nonnegative integers and every $a_i \in \{0, 1\}$. The sequence is traditionally written in reverse order:

$\ldots, a_4, a_3, a_2, a_1, a_0$

The unique sequence corresponding to an integer x is generated by the formula

$a_i = \lfloor x / 2^i \rfloor \textbf{ mod } 2$

~~Zero~~ If *x* is zero or ~~a~~ positive ~~integer is interpreted as having~~, then its sequence will have infinitely many consecutive leading 0~~'~~'s ~~as its most significant bits~~, while a negative integer *x* will generate a sequence with ~~is interpreted as having~~ infinitely many consecutive leading ~~1's~~ 1'sas ~~its most significant bits~~. For example, 6 ~~is interpreted~~ generates the sequence ~~as~~ ...0...0000110, while –6 ~~is interpreted as~~generates ...1...1111010.

The logical AND, OR, and XOR operations below operate on corresponding elements of the sequences a_i and b_i generated by the two parameters *x* and *y*. The result is another infinite sequence of bits c_i. The result of the operation is the unique integer *z* that generates the sequence c_i. For example, ~~;~~ ANDing ~~them together~~corresponding elements of the sequences generated by 6 and –6 yields the sequence ...0...0000010, which is the sequence generated by the integer 2. Thus, *bitwiseAnd*(6, –6) = 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 $\lfloor x \times 2^{count} \rfloor$.

5.7 Floating-Point Numbers

The semantic domain FLOAT64 is comprised of all nonzero rational numbers representable as double-precision floating-point IEEE 754 values, together with five special tags **+zero**, **–zero**, **+∞**, **–∞**, and **NaN**. FLOAT64 is the union of the following semantic domains:

FLOAT64 = FINITEFLOAT64 ∪ {**+∞**, **–∞**, **NaN**};
FINITEFLOAT64 = NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64 ∪ {**+zero**, **–zero**};

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

NORMALISEDFLOAT64 = $\{s \times m \times 2^e \mid \forall s \in \{–1, 1\}, \forall m \in \{2^{52} \ldots 2^{53}–1\}, \forall e \in \{–1074 \ldots 971\}\}$

m is called the *significand*.

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

DENORMALISEDFLOAT64 = $\{s \times m \times 2^{–1074} \mid \forall s \in \{–1, 1\}, \forall m \in \{1 \ldots 2^{52}–1\}\}$

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). All not-a-number values are considered indistinguishable from each other.

Members of the semantic domain NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64 that are greater than zero are called *positive finite*. The remaining members of NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64 are less than zero and are called *negative finite*.

Since floating-point numbers are either rational numbers or tags, the notation = and ≠ may be used to compare them. Note that = is **false** for different tags, so **+zero** ≠ **–zero** but **NaN** = **NaN**. The ECMAScript *x* == *y* and *x* === *y* operators have different behaviour for floating-point numbers, defined as *float64Compare*(*x*, *y*) = **equal**.

5.7.1 Conversion

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

proc *realToFloat64*(*x*: REAL): FLOAT64
  s: RATIONAL{} ← NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64 ∪ {$–2^{1024}$, 0, 2^{1024}};
  Let *a*: RATIONAL 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 $–2^{1024}$, 0, and 2^{1024} are considered to have even significands.
  if *a* = 2^{1024} **then return +∞**
  elsif *a* = $–(2^{1024})$ **then return –∞**
  elsif *a* ≠ 0 **then return** *a*
  elsif *x* < 0 **then return –zero**
  else return +zero
  end if
end proc

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

The procedure *truncateFiniteFloat64* truncates a FINITEFLOAT64 value to an integer, rounding towards zero:

proc *truncateFiniteFloat64*(*x*: FINITEFLOAT64): INTEGER
  if *x* ∈ {**+zero**, **–zero**} **then return** 0 **end if**;
  if *x* > 0 **then return** $\lfloor x \rfloor$ **else return** $\lceil x \rceil$ **end if**
end proc

5.7.2 Comparison

ORDER is the four-element semantic domain of tags representing the possible results of a floating-point comparison:

ORDER = {**less**, **equal**, **greater**, **unordered**}

The procedure *rationalCompare* compares two rational values *x* and *y* and returns one of the tags **less**, **equal**, or **greater** depending on the result of the comparison:

proc *rationalCompare*(*x*: RATIONAL, *y*: RATIONAL): ORDER
  if *x* < *y* **then return less**
  elsif *x* = *y* **then return equal**
  else return greater
  end if
end proc

The procedure *float64Compare* compares two FLOAT64 values *x* and *y* and returns one of the tags **less**, **equal**, **greater**, or **unordered** depending on the result of the comparison according to the table below.

float64Compare(*x*: FLOAT64, *y*: FLOAT64): ORDER

	y						
x	**–∞**	negative finite	**–zero**	**+zero**	positive finite	**+∞**	**NaN**
–∞	**equal**	**less**	**less**	**less**	**less**	**less**	**unordered**
negative finite	**greater**	*rationalCompare*(*x*, *y*)	**less**	**less**	**less**	**less**	**unordered**
–zero	**greater**	**greater**	**equal**	**equal**	**less**	**less**	**unordered**
+zero	**greater**	**greater**	**equal**	**equal**	**less**	**less**	**unordered**
positive finite	**greater**	**greater**	**greater**	**greater**	*rationalCompare*(*x*, *y*)	**less**	**unordered**
+∞	**greater**	**greater**	**greater**	**greater**	**greater**	**equal**	**unordered**
NaN	**unordered**	**unordered**	**unordered**	**unordered**	**unordered**	**unordered**	**unordered**

5.7.3 Arithmetic

The following tables define procedures that perform common arithmetic on FLOAT64 values using IEEE 754 rules. All procedures are strict and evaluate all of their arguments left-to-right.

float64Abs(*x*: FLOAT64): FLOAT64

x	**Result**
−∞	**+∞**
negative finite	–*x*
−zero	**+zero**
+zero	**+zero**
positive finite	*x*
+∞	**+∞**
NaN	**NaN**

float64Negate(*x*: FLOAT64): FLOAT64

x	**Result**
−∞	**+∞**
negative finite	–*x*
−zero	**+zero**
+zero	**−zero**
positive finite	–*x*
+∞	**−∞**
NaN	**NaN**

float64Add(*x*: FLOAT64, *y*: FLOAT64): FLOAT64

	y						
x	**−∞**	negative finite	**−zero**	**+zero**	positive finite	**+∞**	**NaN**
−∞	**−∞**	**−∞**	**−∞**	**−∞**	**−∞**	**NaN**	**NaN**
negative finite	**−∞**	*realToFloat64*(*x* + *y*)	*x*	*x*	*realToFloat64*(*x* + *y*)	**+∞**	**NaN**
−zero	**−∞**	*y*	**−zero**	**+zero**	*y*	**+∞**	**NaN**
+zero	**−∞**	*y*	**+zero**	**+zero**	*y*	**+∞**	**NaN**
positive finite	**−∞**	*realToFloat64*(*x* + *y*)	*x*	*x*	*realToFloat64*(*x* + *y*)	**+∞**	**NaN**
+∞	**NaN**	**+∞**	**+∞**	**+∞**	**+∞**	**+∞**	**NaN**
NaN	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**

NOTE The identity for floating-point addition is **−zero**, not **+zero**.

float64Subtract(*x*: FLOAT64, *y*: FLOAT64): FLOAT64

	y						
x	**−∞**	negative finite	**−zero**	**+zero**	positive finite	**+∞**	**NaN**
−∞	**NaN**	**−∞**	**−∞**	**−∞**	**−∞**	**−∞**	**NaN**
negative finite	**+∞**	*realToFloat64*(*x* – *y*)	*x*	*x*	*realToFloat64*(*x* – *y*)	**−∞**	**NaN**
−zero	**+∞**	–*y*	**+zero**	**−zero**	–*y*	**−∞**	**NaN**
+zero	**+∞**	–*y*	**+zero**	**+zero**	–*y*	**−∞**	**NaN**
positive finite	**+∞**	*realToFloat64*(*x* – *y*)	*x*	*x*	*realToFloat64*(*x* – *y*)	**−∞**	**NaN**
+∞	**+∞**	**+∞**	**+∞**	**+∞**	**+∞**	**NaN**	**NaN**
NaN	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**	**NaN**

float64Multiply(*x*: FLOAT64, *y*: FLOAT64): FLOAT64

x \ *y*	−∞	negative finite	−zero	+zero	positive finite	+∞	NaN
−∞	+∞	+∞	NaN	NaN	−∞	−∞	NaN
negative finite	+∞	*realToFloat64*(*x* × *y*)	+zero	−zero	*realToFloat64*(*x* × *y*)	−∞	NaN
−zero	NaN	+zero	+zero	−zero	−zero	NaN	NaN
+zero	NaN	−zero	−zero	+zero	+zero	NaN	NaN
positive finite	−∞	*realToFloat64*(*x* × *y*)	−zero	+zero	*realToFloat64*(*x* × *y*)	+∞	NaN
+∞	−∞	−∞	NaN	NaN	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

float64Divide(*x*: FLOAT64, *y*: FLOAT64): FLOAT64

x \ *y*	−∞	negative finite	−zero	+zero	positive finite	+∞	NaN
−∞	NaN	+∞	+∞	−∞	−∞	NaN	NaN
negative finite	+zero	*realToFloat64*(*x* / *y*)	+∞	−∞	*realToFloat64*(*x* / *y*)	−zero	NaN
−zero	+zero	+zero	NaN	NaN	−zero	−zero	NaN
+zero	−zero	−zero	NaN	NaN	+zero	+zero	NaN
positive finite	−zero	*realToFloat64*(*x* / *y*)	−∞	+∞	*realToFloat64*(*x* / *y*)	+zero	NaN
+∞	NaN	−∞	−∞	+∞	+∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

float64Remainder(*x*: FLOAT64, *y*: FLOAT64): FLOAT64

x \ *y*	−∞	negative finite	−zero	+zero	positive finite	+∞	NaN
−∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
negative finite	*x*	*float64Negate*(*float64Remainder*(–*x*, –*y*))	NaN	NaN	*float64Negate*(*float64Remainder*(–*x*, *y*))	*x*	NaN
−zero	−zero	−zero	NaN	NaN	−zero	−zero	NaN
+zero	+zero	+zero	NaN	NaN	+zero	+zero	NaN
positive finite	*x*	*float64Remainder*(*x*, –*y*)	NaN	NaN	*realToFloat64*(*x* – *y*×⌊*x*/*y*⌋)	*x*	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

~~5.7~~5.8 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 semantic domain~~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~~5.9 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.

A list can also be written using the list comprehension notation

$\mathbf{[}f(x) \mid \forall x \in u\mathbf{]}$

which denotes the list $\mathbf{[}f(u[0]), f(u[1]), \ldots , f(u[|u|-1])\mathbf{]}$ whose elements consist of the results of applying expression f to each corresponding element of list u. x is the name of the parameter in expression f. A predicate can be added:

$\mathbf{[}f(x) \mid \forall x \in u$ **such that** $predicate(x)\mathbf{]}$

denotes the list of the results of computing expression f on all elements x of list u that satisfy the $predicate$ expression. The results are listed in the same order as the elements x of list u. For example,

$\mathbf{[}x^2 \mid \forall x \in \mathbf{[}-1, 1, 2, 3, 4, 2, 5\mathbf{]]} = \mathbf{[}1, 1, 4, 9, 16, 4, 25\mathbf{]}$

$\mathbf{[}x+1 \mid \forall x \in \mathbf{[}-1, 1, 2, 3, 4, 5, 3, 10\mathbf{]}$ **such that** x **mod** $2 = 1\mathbf{]} = \mathbf{[}0, 2, 4, 6, 4\mathbf{]}$

Let $u = \mathbf{[}e_0, e_1, \ldots , e_{n-1}\mathbf{]}$ and $v = \mathbf{[}f_0, f_1, \ldots , f_{m-1}\mathbf{]}$ 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
$\|u\|$		The length n of the list
$u[i]$	$0 \le i < \|u\|$	The i^{th} element e_i.
$u[i \ldots j]$	$0 \le i \le j+1 \le \|u\|$	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 $j=i-1$.
$u[i \ldots]$	$0 \le i \le \|u\|$	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 $i=n$.
$u[i \setminus x]$	$0 \le i < \|u\|$	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
$u \oplus v$		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 semantic domain~~set~~, then T[] is the semantic domain~~set~~ of all lists whose elements are members of T. The empty list **[]** is a member of T[] for any semantic domain~~set~~ T.

In addition to the above, the **some** and **every** quantifiers can be used on lists just as on sets:

some $x \in u$ **satisfies** $predicate(x)$

every $x \in u$ **satisfies** $predicate(x)$

These quantifiers' behaviour on lists is analogous to that on sets, except that, if the **some** quantifier returns **true** then it leaves variable x set to the *first* element of list u that satisfies condition $predicate(x)$. For example,

some $x \in \mathbf{[}3, 36, 19, 26\mathbf{]}$ **satisfies** x **mod** $10 = 6$

evaluates to **true** and leaves x set to 36.

~~5.9~~5.10 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 semantic domain~~set~~ of all strings. STRING = CHARACTER[].

~~5.10~~5.11 Tuples

A *tuple* is an immutable aggregate of values comprised of a name NAME 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 semantic domain~~set~~s of possible values that the corresponding fields may hold.

The notation

$\text{NAME}\langle v_1, \ldots, v_n\rangle$

represents a tuple with name 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 semantic domain~~set~~ T_i.

If *a* is the tuple $\text{NAME}\langle v_1, \ldots, v_n\rangle$, then

$a.\text{label}_i$

returns the i^{th} field’s value v_i.

~~When used in an expression, the tuple’s name NAME itself represents the set of all tuples with name NAME.~~

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

When used in an expression, the tuple’s name NAME itself represents the semantic domain of all tuples with name NAME.

~~5.11~~5.12 Records

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

A record is comprised of a name NAME 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 semantic domain~~set~~s of possible values that the corresponding fields may hold.

The expression

new NAME⟪v_1, ... , v_n⟫

creates a record with name 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 semantic domain~~set~~ T_i.

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

$a.\mathsf{label}_i$

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 semantic domain~~set~~ T_i, using the assignment

$a.\mathsf{label}_i \leftarrow w$

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

~~When used in an expression, the record's name NAME itself represents the set of all records with name NAME.~~

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

When used in an expression, the record's name NAME itself represents the semantic domain of all records with name NAME.

~~5.12~~5.13 Procedures

A procedure is a function that receives zero or more arguments, performs computations, and optionally returns a result. Procedures may perform side effects. In this document the word *procedure* is used to refer to internal algorithms; the word *function* is used to refer to the programmer-visible `function` ECMAScript construct.

A procedure is denoted as:

proc $f(param_1: T_1, \ldots, param_n: T_n): T$
  $step_1$;
  $step_2$;
  ... ;
  $step_m$
end proc;

If the procedure does not return a value, the : T on the first line is omitted.

f is the procedure's name, $param_1$ through $param_n$ are the procedure's parameters, T_1 through T_n are the parameters' respective ~~constraint set~~semantic domains, T is the semantic domain~~constraint set~~ of the procedure's result, and $step_1$ through $step_m$ describe the procedure's computation steps, which may produce side effects and/or return a result. If T is omitted, the procedure does not return a result. When the procedure is called with argument values v_1 through v_n, the procedure's steps are performed and the result, if any, returned to the caller.

A procedure's steps can refer to the parameters $param_1$ through $param_n$; each reference to a parameter $param_i$ evaluates to the corresponding argument value v_i. Procedure parameters are statically scoped. Arguments are passed by value.

~~For convenience, if the procedure's body is comprised of only a **return** step, the procedure~~

~~**proc** $f(param_1: T_1, \ldots, param_n: T_n): T$~~
  ~~**return** *expression*~~
~~**end proc**;~~

~~is abbreviated as:~~

~~**proc** $f(param_1: T_1, \ldots, param_n: T_n): T \equiv expression$~~

~~5.12.1~~5.13.1 Operations

The only operation done on a procedure *f* is calling it using the $f(arg_1, \ldots, arg_n)$ syntax. *f* is computed first, followed by the argument expressions arg_1 through arg_n, in left-to-right order. If the result of computing *f* or any of the argument expressions throws an exception *e*, then the call immediately propagates *e* without computing any following argument expressions. Otherwise, *f* is invoked using the provided arguments and the resulting value, if any, returned to the caller.

Procedures are never compared using =, ≠, or any of the other comparison operators.

~~5.12.2~~5.13.2 Semantic Domain~~Set~~s of Procedures

The semantic domain~~set~~ of procedures that take *n* parameters ~~with constraints~~in semantic domains T_1 through T_n respectively and produce a result ~~with constraint~~in semantic domain T is written as $T_1 \times T_2 \times \ldots \times T_n \to T$. If $n = 0$, this semantic domain~~set~~ is written as $() \to T$. If the procedure does not produce a result, the semantic domain~~set~~ of procedures is written either as $T_1 \times T_2 \times \ldots \times T_n \to ()$ or as $() \to ()$.

~~To avoid set-theoretical paradoxes, these sets only include procedures that are present in the semantics or derived from them in the standard domain-theoretical manner.~~

~~5.12.3~~5.13.3 Steps

Computation steps in procedures 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.

nothing

A **nothing** step performs no operation.

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 or mutable global (see *****) *v*. If this is the first time ~~the~~ a temporary variable is referenced in a procedure, the variable's semantic domain~~constraint set~~ T is listed; any value stored in *v* is guaranteed to be a member of the semantic domain~~set~~ T.

Temporary variables are local to the procedures that define them (including any nested procedures). Each time a procedure 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**
T_1~~set_1~~ **do** *step*; *step*; ...; *step*;
T_2~~set_2~~ **do** *step*; *step*; ...; *step*;
...;
T_n~~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 $v \in T_1$~~set_1~~, then the first list of *step*s is performed. Otherwise, if $v \in T_2$~~set_2~~, then the second list of *step*s is performed, and so on. If *v* is not a member of any T_i~~*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 T_i~~*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 procedure 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 procedure with the result *v*. No further steps in the enclosing procedure are performed. The *expression* may be omitted, in which case the enclosing procedure 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 procedure calls until the exception is caught by a **catch** step. Unless the enclosing procedure catches this exception, no further steps in the enclosing procedure are performed.

try
step;
step;
…;
step
catch *v*: T~~*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 procedure), then the **try** step is done. If any of the *step*s propagates out an exception *e*, then if *e* ∈ T~~*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* ∉ T~~*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.

~~5.12.4~~5.13.4 Nested Procedures

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

~~5.13~~5.14 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~~5.14.1 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*: 12.1)
  | *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~~5.14.2 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~~5.14.3 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~~5.14.4 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~~5.14.5 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 *****) 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`, `named`, `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~~ number'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 *****). **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~~ analyses 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 5.14). Use the start symbol *NextInputElement*^{re}, *NextInputElement*^{div}, or *NextInputElement*^{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^{re} ⇒ *WhiteSpace* *InputElement*^{re} (*WhiteSpace*: 7.2)

NextInputElement^{div} ⇒ *WhiteSpace* *InputElement*^{div}

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

~~| _ *IdentifierName*~~

InputElement^{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^{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*]
Return a **string** token with string contents *LexString*[*IdentifierName*].

Lex[*InputElement*^{ν} ⇒ *LineBreaks*] = **lineBreak**

Lex[*InputElement*^{ν} ⇒ *IdentifierOrKeyword*] = *Lex*[*IdentifierOrKeyword*]

Lex[*InputElement*^{ν} ⇒ *Punctuator*] = *Lex*[*Punctuator*]

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

Lex[*InputElement*^{ν} ⇒ *NumericLiteral*] = *Lex*[*NumericLiteral*]

Lex[*InputElement*^{ν} ⇒ *StringLiteral*] = *Lex*[*StringLiteral*]

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

Lex[*InputElement*^{ν} ⇒ *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 *****).

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`, `named`, `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`, `named`, 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*$^{\text{single}}$ `'`
| `"` *StringChars*$^{\text{double}}$ `"`

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

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

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

LiteralStringChar$^{\text{double}}$ ⇒ *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*$^{\text{single}}$ `'`]
Return a **string** token with string contents *LexString*[*StringChars*$^{\text{single}}$].

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

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 implementations are encouraged to implement these objects in ~~other~~ more efficient 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~~Boolean, a number, a string, a namespace, an attribute, a class, a method closure, a prototype instance, or a ~~general~~ class instance. These kinds of objects are described in the subsections below.

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

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

9.1.1 Undefined

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

UNDEFINED = {**undefined**}

9.1.2 Null

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

NULL = {**null**}

9.1.3 Booleans

There are two ~~booleans~~Booleans, **true** and **false**. The semantic domain~~set~~ BOOLEAN consists of these two values. See section 5.4.

9.1.4 Numbers

The semantic domain~~set~~ FLOAT64 consists of all representable double-precision floating-point IEEE 754 values. See section 5.7.

9.1.5 Strings

The semantic domain~~set~~ STRING consists of all representable strings. See section 5.10. 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 NAMESPACE record (see section 5.12) with 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.

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 an ATTRIBUTE tuple (see section 5.11) with the fields below.

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
dynamic~~classMod~~	BOOLEAN~~CLASSMODIFIER~~	**true** if the `dynamic` attribute has been given~~**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; ... if not. MEMBERMODIFIER = {

		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 CLASS records (see section 5.12) with the fields below.

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
dynamic~~classMod~~	BOOLEAN~~CLASSMODIFIER~~	**true** if this class or any of its ancestors was defined with the `dynamic` attribute~~**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 procedure to call (see section 9.6) when this class is used in a call expression
construct	INVOKER	A procedure to call (see section 9.6) when this class is used in a `new` expression

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

CLASSOPT = NULL ∪ CLASS

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 GLOBALMEMBER record (see section 5.12) has the fields below and controls the behaviour of either reading or writing a global property of ~~an instance of~~ a class.

Field	Contents	Note
name	STRING	The member's unqualified name
namespaces	NAMESPACE{}	The set of namespaces qualifying name. This set is never empty.
access	MEMBERACCESS	Describes whether this member is read-only, write-only, or read-write
category	GLOBALCATEGORY	The member's category. GLOBALCATEGORY = {**static**, **constructor**}

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~~holds the value of this member (and specifies that this member is a field); 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*.

An INSTANCEMEMBER record (see section 5.12) has the fields below and controls the behaviour of either reading or writing a property of an instance of a class.

Field	Contents	Note
name	STRING	The member's unqualified name
namespaces	NAMESPACE{}	The set of namespaces qualifying name. This set is never empty.
access	MEMBERACCESS	Describes whether this member is read-only, write-only, or read-write
category	INSTANCECATEGORY	The member's category. INSTANCECATEGORY = {**abstract**, **virtual**, **final**}
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 SLOT that holds the value of this member in an instance (and specifies that this member is a field)~~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 semantic domain~~set~~s are unions of their instance and global equivalents:

MEMBER = INSTANCEMEMBER ∪ GLOBALMEMBER;

MEMBERDATA = INSTANCEDATA ∪ GLOBALDATA;

MEMBERDATAOPT = NULL ∪ MEMBERDATA

MEMBERACCESS = {**read**, **write**, **readWrite**};

The MEMBERACCESS semantic domain describes whether a member is read-only, write-only, or read-write. There can be two separate members with the same name in the same object only if one of them is read-only and the other write-only.

A GLOBALSLOT record (see section 5.12) has the fields below and holds the type and value of a class-global property of a class.

Field	Contents	Note
type	CLASS	The type of values that can be stored in this slot
value	OBJECT	This class-global property's current value

A METHOD record (see section 5.12) has the fields below and describes a non-accessor member defined with the `function` keyword.

Field	Contents	Note
type	SIGNATURE	The method's signature (see 9.5)
f	INSTANCEOPT	A callable object or **null** if this is an abstract method

An ACCESSOR record (see section 5.12) has the fields below and describes an accessor — a member defined with the `function get` or `function set` keywords that runs code to do the read or write.

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 METHODCLOSURE tuple (see section 5.11) has the fields below and describes an instance method with a bound `this` value.

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

9.1.10 Prototype Instances

Prototype instances are represented as PROTOTYPE records (see section 5.12) with the fields below. Prototype instances contain no fixed properties.

Field	Contents	Note
parent	PROTOTYPEOPT	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.
dynamicProperties	DYNAMICPROPERTY{}	A set of this instance's dynamic properties

PROTOTYPEOPT consists of all PROTOTYPE records as well as **null**:

PROTOTYPEOPT = NULL ∪ PROTOTYPE

A DYNAMICPROPERTY record (see section 5.12) has the fields below and describes one dynamic property of one (prototype or class) instance.

Field	Contents	Note
name	STRING	This dynamic property's name
value	OBJECT	This dynamic property's current value

~~9.1.10~~9.1.11 ~~General~~ Class Instances

Instances of programmer-defined classes as well as of some built-in classes have the semantic domain INSTANCE. If the class of an instance or one of its ancestors has the `dynamic` attribute, then the instance is a DYNAMICINSTANCE record; otherwise, it is a FIXEDINSTANCE record.

INSTANCE = FIXEDINSTANCE ∪ DYNAMICINSTANCE;

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

INSTANCEOPT = NULL ∪ INSTANCE

~~are represented as INSTANCE records (see section 5.12) with the fields below.~~

NOTE Instances of some built-in classes are represented as described in sections 9.1.1 through ~~9.1.9~~9.1.10 rather than as INSTANCE records. This distinction is made for convenience in specifying the language's behaviour and is invisible to the programmer.

Instances of non-`dynamic` classes are represented as FIXEDINSTANCE records (see section 5.12) with the fields below. These instances can contain only fixed properties.

Field	Contents	Note
type	CLASS	This instance's type
call	INVOKER	A procedure to call when this instance is used in a call expression
construct	INVOKER	A procedure 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

Instances of `dynamic` classes are represented as DYNAMICINSTANCE records (see section 5.12) with the fields below. These instances can contain fixed and dynamic properties.

Field	Contents	Note
type	CLASS	This instance's type
call	INVOKER	A procedure to call when this instance is used in a call expression
construct	INVOKER	A procedure 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

9.1.11.1 Slots

A SLOT record (see section 5.12) has the fields below and describes the value of one fixed property of one instance.

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

A SLOTID record (see section 5.12) has the field below and serves as a unique identifier that distinguishes one member's slots from another member's.

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

9.2 Qualified Names

A QUALIFIEDNAME tuple (see section 5.11) has the fields below and represents a fully qualified name.

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

A PARTIALNAME tuple (see section 5.11) has the fields below and represents a partially qualified name. A partially qualified name may not have a unique namespace qualifier; rather, it has a set of namespaces any of which could qualify the name.

Field	Contents	Note
namespaces	NAMESPACE{}	A nonempty set of namespaces that may qualify the name
name	STRING	The name

9.3 Objects with Limits

A LIMITEDINSTANCE tuple (see section 5.11) represents an intermediate result of a `super` or `super(`*expr*`)` subexpression. It has the fields below.

Field	Contents	Note
instance	INSTANCE	The value of *expr* to which the `super` subexpression was applied; if *expr* wasn't given, defaults to the value of `this`. The value of instance is always an instance of the limit class or one of its descendants.
limit	CLASS	The class inside which the `super` subexpression was applied

Member and operator lookups on a LIMITEDINSTANCE value will only find members and operators defined on proper ancestors of limit.

OBJOPTIONALLIMIT is the result of a subexpression that can produce either an OBJECT or a LIMITEDINSTANCE:

OBJOPTIONALLIMIT = OBJECT ∪ LIMITEDINSTANCE

~~9.3~~9.4 References

A REFERENCE (also known as an *lvalue* in the computer literature) ~~reference~~ is a temporary result of evaluating ~~many~~ some subexpressions. It is ~~either an OBJECT (also known as an *rvalue* in the computer literature) or~~ a place where a value may be read or written ~~(also known as an *lvalue*)~~. A REFERENCE may serve as either the source or destination of an assignment.

Some subexpressions evaluate to an OBJORREF, which is either an OBJECT (also known as an *rvalue*) or a REFERENCE. Attempting to ~~write to~~use a~~ reference~~n OBJORREF that is an rvalue as the destination of an assignment produces an error.

REFERENCE = DOTREFERENCE ∪ BRACKETREFERENCE;
OBJORREF = OBJECT ∪ REFERENCE

~~REFERENCE = OBJECT ∪ DOTREFERENCE ∪ BRACKETREFERENCE~~

A DOTREFERENCE tuple (see section 5.11) has the fields below and represents an lvalue that refers to a property of the base object with the given partially qualified name. DOTREFERENCE tuples arise from evaluating subexpressions such as *a*`.`*b* or *a*`.`*q*`::`*b*.

Field	Contents	Note
base	OBJOPTIONALLIMIT~~OBJECT~~	The object whose property was referenced (*a* in the examples above). The object may be a LIMITEDINSTANCE if *a* is a `super` expression, in which case the property lookup will be restricted to members defined in proper ancestors of base.limit.
propName	PARTIALNAME	The partially qualified name (*b* or *q*`::`*b* in the examples above)

A BRACKETREFERENCE tuple (see section 5.11) has the fields below and represents an lvalue that refers to the result of applying the `[]` operator to the base object with the given arguments. BRACKETREFERENCE tuples arise from evaluating subexpressions such as *a*`[`*x*`]` or *a*`[`*x*`,`*y*`]`.

Field	Contents	Note
base	OBJOPTIONALLIMIT~~OBJECT~~	The object whose property was referenced (*a* in the examples above). The object may be a LIMITEDINSTANCE if *a* is a `super` expression, in which case the property lookup will be restricted to definitions of the `[]` operator defined in proper ancestors of base.limit.
args	ARGUMENTLIST	The list of arguments between the brackets (*x* or *x*`,`*y* in the examples above)

9.4.1 References with Limits

Some subexpressions evaluate to references with limits. A LIMITEDOBJORREF tuple (see section 5.11) represents an intermediate result of a `super` or `super(`*expr*`)` subexpression in cases where *expr* might be a reference. It has the fields below.

Field	Contents	Note
ref	OBJORREF	The value of *expr* to which the `super` subexpression was applied; if *expr* wasn't given, defaults to the value of `this`
limit	CLASS	The class inside which the `super` subexpression was applied

The algorithms in the later chapters first convert a LIMITEDOBJORREF tuple into a LIMITEDINSTANCE tuple (see section 9.3) before operating on it.

~~9.4~~9.5 Signatures

A SIGNATURE tuple (see section 5.11) has the fields below and represents the type signature of a function.

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
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 NAMEDPARAMETER tuple (see section 5.11) has the fields below and represents the signature of one named parameter.

Field	Contents	Note
name	STRING	This parameter's name
type	CLASS	This parameter's type

~~9.5~~9.6 Argument Lists

An ARGUMENTLIST tuple (see section 5.11) has the fields below and describes the arguments (other than `this`) passed to a function.

Field	Contents	Note
positional	OBJECT[]	Ordered list of positional arguments
named	NAMEDARGUMENT{}	Set of named arguments

A NAMEDARGUMENT tuple (see section 5.11) has the fields below and describes one named argument passed to a function.

Field	Contents	Note
name	STRING	This argument's name
value	OBJECT	This argument's value

INVOKER is the semantic domain~~set~~ of procedures that take an OBJECT (the `this` value) and an ARGUMENTLIST and produce an OBJECT result.

INVOKER = OBJECT × ARGUMENTLIST → OBJECT

~~9.6~~9.7 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 held in a mutable global variable that contains a UNARYMETHOD{} set of defined unary methods.

~~an independent UNARYTABLE record (see section 5.11) with the field below.~~

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

A UNARYMETHOD tuple (see section 5.11) has the fields below and represents one unary operator method.

Field	Contents	Note
operandType	CLASS	The dispatched operand’s type
~~op~~f	OBJECT × OBJECT × ARGUMENTLIST → OBJECT	Procedure that takes a `this` value, a first positional argument, and an ARGUMENTLIST of other positional and named arguments and returns the operator’s result

~~9.7~~9.8 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 held in a mutable global variable that contains a BINARYMETHOD{} set of defined binary methods.~~an independent BINARYTABLE record (see section 5.11) with the field below.~~

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

A BINARYMETHOD tuple (see section 5.11) has the fields below and represents one binary operator method.

Field	Contents	Note
leftType	CLASS	The left operand’s type
rightType	CLASS	The right operand’s type
~~op~~f	OBJECT × OBJECT → OBJECT	Procedure 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 Numeric Utilities

proc *uInt32ToInt32*(*i*: INTEGER): INTEGER
  if $i < 2^{31}$ **then return** *i* **else return** $i - 2^{32}$ **end if**
end proc;

proc *toUInt32*(*x*: FLOAT64): INTEGER
  if *x* ∈ {**+∞**, **−∞**, **NaN**} **then return** 0 **end if**;
  return *truncateFiniteFloat64*(*x*) **mod** 2^{32}
end proc;

proc *toInt32*(*x*: FLOAT64): INTEGER
  return *uInt32ToInt32*(*toUInt32*(*x*))
end proc;

~~10.1~~10.2 Object Utilities

~~10.1.1~~10.2.1 *objectType*

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

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

~~10.1.2~~10.2.2 *hasType*

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

hasType(*o*, *c*) returns **true** if *o* is an instance of class *c* (or one of *c*'s subclasses). It considers **null** to be an instance of the classes `Null` and `Object` only.

proc *hasType*(*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
end proc

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

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

~~10.1.3~~10.2.3 *toBoolean*

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

proc *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 ∪ PROTOTYPE **do return true**;
    INSTANCE **do** ????
  end case
end proc;

~~10.1.4~~10.2.4 *toNumber*

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

proc *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**;
    PROTOTYPE ∪ INSTANCE **do** ????
  end case
end proc;

~~10.1.5~~10.2.5 *toString*

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

proc *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** ????;
    PROTOTYPE ∪ INSTANCE **do** ????
  end case
end proc;

~~10.1.6~~10.2.6 *unaryPlus*

unaryPlus(*o*) returns the value of the unary expression +*o*.

proc *unaryPlus*(*a*: ObjOptionalLimit~~Object~~): Object
  return *unaryDispatch*(*plusTable*, **null**, ~~**null**,~~ *a*, ArgumentList⟨[], {}⟩)
end proc;

~~10.1.7~~10.2.7 *unaryNot*

unaryNot(*o*) returns the value of the unary expression !*o*.

proc *unaryNot*(*a*: Object): Object
  return not *toBoolean*(*a*)
end proc;

10.3 Objects with Limits

getObject(*o*) returns *o* without its limit, if any.

proc *getObject*(*o*: ObjOptionalLimit): Object
  case *o* **of**
    Object **do return** *o*;
    LimitedInstance **do return** *o*.instance
  end case
end proc;

getObjectLimit(*o*) returns *o*'s limit or **null** if none is provided.

proc *getObjectLimit*(*o*: ObjOptionalLimit): ClassOpt
  case *o* **of**
    Object **do return null**;
    LimitedInstance **do return** *o*.limit
  end case
end proc;

~~10.2~~10.4 References

If *r* is an Object, *readReference*(*r*) returns it unchanged. If *r* is a Reference, this function reads *r* and returns the result.

proc *readReference*(*r*: ObjOrRef): Object
  case *r* **of**
    Object **do return** *r*;
    DotReference **do return** *readProperty*(*r*.base, *r*.propName);
    BracketReference **do return** *unaryDispatch*(*bracketReadTable*, **null**, *r*.base, *r*.args)
  end case
end proc;

readRefWithLimit(*r*) reads the reference, if any, inside *r* and returns the result, retaining the same limit as *r*. If *r* has a limit *limit*, then the object read from the reference is checked to make sure that it is an instance of *limit* or one of its descendants.

proc *readRefWithLimit*(*r*: ObjOrRefOptionalLimit): ObjOptionalLimit
  case *r* **of**
    ObjOrRef **do return** *readReference*(*r*);
    LimitedObjOrRef **do**
      o: Object ← *readReference*(*r*.ref);
      limit: Class ← *r*.limit;
      if *o* = **null then return null end if**;
      if *o* ∉ Instance **or not** *hasType*(*o*, *limit*) **then throw typeError end if**;
      return LimitedInstance⟨*o*, *limit*⟩
  end case
end proc;

If *r* is a reference, *writeReference*(*r*, *o*) writes *o* into *r*. An error occurs if *r* is not a reference. *r*'s limit, if any, is ignored.

proc *writeReference*(*r*: OBJORREFOPTIONALLIMIT, *o*: OBJECT)
  case *r* **of**
    OBJECT **do throw referenceError**;
    DOTREFERENCE **do** *writeProperty*(*r*.base, *r*.propName, *o*);
    BRACKETREFERENCE **do**
      args: ARGUMENTLIST ← ARGUMENTLIST⟨**[***o***]** ⊕ *r*.args.positional, *r*.args.named⟩;
      unaryDispatch(*bracketWriteTable*, **null**, *r*.base, *args*);
    LIMITEDOBJORREF **do** *writeReference*(*r*.ref, *o*)
  end case
end proc;

If *r* is a REFERENCE, *deleteReference*(*r*) deletes it. If *r* is an OBJECT, this function signals an error.

proc *deleteReference*(*r*: OBJORREF): OBJECT
  case *r* **of**
    OBJECT **do throw referenceError**;
    DOTREFERENCE **do return** *deleteProperty*(*r*.base, *r*.propName);
    BRACKETREFERENCE **do**
      return *unaryDispatch*(*bracketDeleteTable*, **null**, *r*.base, *r*.args)
  end case
end proc;

referenceBase(*r*) returns REFERENCE *r*'s base or**null** if there is none. *r*'s limit and the base's limit, if any, are ignored.

proc *referenceBase*(*r*: OBJORREFOPTIONALLIMIT): OBJECT
  case *r* **of**
    OBJECT **do return null**;
    REFERENCE **do return** *getObject*(*r*.base);
    LIMITEDOBJORREF **do return** *referenceBase*(*r*.ref)
  end case
end proc;

~~Read the OBJORREF *r*.~~

~~**proc** *readReference*(*r*: OBJORREF): OBJECT~~
  ~~**case** *r* **of**~~
    ~~OBJECT **do return** *r*;~~
    ~~DOTREFERENCE **do return** *readProperty*(*r*.base, *r*.propName, *r*.super);~~
    ~~BRACKETREFERENCE **do**~~
      ~~**return** *unaryDispatch*(*bracketReadTable*, *r*.super, **null**, *r*.base, *r*.args)~~
  ~~**end case**~~
~~**end proc**;~~

~~Write *o* into the OBJORREF *r*.~~

~~**proc** *writeReference*(*r*: OBJORREF, *o*: OBJECT)~~
  ~~**case** *r* **of**~~
    ~~OBJECT **do throw referenceError**;~~
    ~~DOTREFERENCE **do** *writeProperty*(*r*.base, *r*.propName, *r*.super, *o*);~~
    ~~BRACKETREFERENCE **do**~~
      ~~*args*: ARGUMENTLIST ← ARGUMENTLIST⟨**[***o***]** ⊕ *r*.args.positional, *r*.args.named⟩;~~
      ~~*unaryDispatch*(*bracketWriteTable*, *r*.super, **null**, *r*.base, *args*)~~
  ~~**end case**~~
~~**end proc**;~~

~~**proc** *deleteReference*(*r*: OBJORREF): OBJECT~~
~~**case** *r* **of**~~
~~OBJECT **do throw referenceError**;~~
~~DOTREFERENCE **do return** *deleteProperty*(*r*.base, *r*.propName, *r*.super);~~
~~BRACKETREFERENCE **do**~~
~~**return** *unaryDispatch*(*bracketDeleteTable*, *r*.super, **null**, *r*.base, *r*.args)~~
~~**end case**~~
~~**end proc**;~~

~~**proc** *referenceBase*(*r*: OBJORREF): OBJECT~~
~~**case** *r* **of**~~
~~OBJECT **do return null**;~~
~~REFERENCE **do return** *r*.base~~
~~**end case**~~
~~**end proc**;~~

~~10.3~~10.5 Member Lookup

10.5.1 Reading a Property

readProperty(*ol*, *pn*) reads the property *pn* of object *o* and returns the value of the property. *readProperty* works by calling *resolveObjectNamespace* to find the right namespace and then reads the fully qualified property.

proc *readProperty*(*ol*: OBJOPTIONALLIMIT, *pn*: PARTIALNAME): OBJECT
ns: NAMESPACE ← *resolveObjectNamespace*(*getObject*(*ol*), *pn*, {**read**, **readWrite**});
qn: QUALIFIEDNAME ← QUALIFIEDNAME⟨*ns*, *pn*.name⟩;
return *readQualifiedProperty*(*ol*, *qn*, **false**)
end proc;

readQualifiedProperty(*ol*, *qn*, *indexableOnly*) reads the property *qn* of object *o* and returns the value of the property. If *indexableOnly* is **true**, only `indexable` properties are considered. *qn*'s namespace must be `public` if *indexableOnly* is **true**.

```
proc readQualifiedProperty(ol: ObjOptionalLimit, qn: QualifiedName, indexableOnly: Boolean): Object
  d: MemberDataOpt ← null;
  case ol of
    Undefined ∪ Null ∪ Boolean ∪ Float64 ∪ String ∪ Namespace ∪ Attribute ∪ MethodClosure ∪
        FixedInstance do
      d ← mostSpecificMember(objectType(ol), false, qn, {read, readWrite}, indexableOnly);
    Class do d ← mostSpecificMember(ol, true, qn, {read, readWrite}, indexableOnly);
    Prototype do
      if qn.namespace ≠ publicNamespace then throw propertyNotFoundError
      elsif some p ∈ ol.dynamicProperties satisfies p.name = qn.name then
        return p.value
      elsif ol.parent = null then return undefined
      else return readQualifiedProperty(ol.parent, qn, indexableOnly)
      end if;
    DynamicInstance do
      d ← mostSpecificMember(objectType(ol), false, qn, {read, readWrite}, indexableOnly);
      if d = null and qn.namespace = publicNamespace then
        if some p ∈ ol.dynamicProperties satisfies p.name = qn.name then
          return p.value
        else return undefined
        end if
      end if;
    LimitedInstance do
      d ← mostSpecificMember(ol.limit.super, false, qn, {read, readWrite}, indexableOnly)
  end case;
  o: Object ← getObject(ol);
  case d of
    {null} do throw propertyNotFoundError;
    GlobalSlot do return d.value;
    SlotId do return findSlot(o, d).value;
    Method do return MethodClosure⟨o, d⟩;
    Accessor do return d.f.call(o, ArgumentList⟨[], {}⟩)
  end case
end proc;
```

10.5.2 Writing a Property

writeProperty(*ol*, *pn*, *newValue*) writes *newValue* into the property *pn* of object *o*. *writeProperty* works by calling *resolveObjectNamespace* to find the right namespace and then writes the fully qualified property.

```
proc writeProperty(ol: ObjOptionalLimit, pn: PartialName, newValue: Object)
  ns: Namespace ← resolveObjectNamespace(getObject(ol), pn, {write, readWrite});
  qn: QualifiedName ← QualifiedName⟨ns, pn.name⟩;
  writeQualifiedProperty(ol, qn, false, newValue)
end proc;
```

writeQualifiedProperty(*ol*, *qn*, *indexableOnly*, *newValue*) writes *newValue* into the property *qn* of object *o*. If *indexableOnly* is **true**, only `indexable` properties are considered. *qn*'s namespace must be `public` if *indexableOnly* is **true**.

```
proc writeQualifiedProperty(ol: ObjOptionalLimit, qn: QualifiedName, indexableOnly: Boolean,
    newValue: Object)
  d: MemberDataOpt ← null;
  case ol of
    Undefined ∪ Null ∪ Boolean ∪ Float64 ∪ String ∪ Namespace ∪ Attribute ∪ MethodClosure do
      throw propertyNotFoundError;
    Class do
      d ← mostSpecificMember(ol, true, qn, {write, readWrite}, indexableOnly);
    Prototype do
      if qn.namespace ≠ publicNamespace then throw propertyNotFoundError end if;
      writeDynamicProperty(ol, qn.name, newValue);
      return;
    FixedInstance do
      d ← mostSpecificMember(objectType(ol), false, qn, {write, readWrite}, indexableOnly);
    DynamicInstance do
      d ← mostSpecificMember(objectType(ol), false, qn, {write, readWrite}, indexableOnly);
      if d = null and qn.namespace = publicNamespace then
        d ← mostSpecificMember(objectType(ol), false, qn, {read, write, readWrite}, indexableOnly);
        if d ≠ null then throw propertyNotFoundError end if;
        writeDynamicProperty(ol, qn.name, newValue);
        return
      end if;
    LimitedInstance do
      d ← mostSpecificMember(ol.limit.super, false, qn, {write, readWrite}, indexableOnly)
  end case;
  o: Object ← getObject(ol);
  case d of
    {null} do throw propertyNotFoundError;
    GlobalSlot do
      if not relaxedHasType(newValue, d.type) then throw typeError end if;
      d.value ← newValue;
    SlotId do
      if not relaxedHasType(newValue, d.type) then throw typeError end if;
      findSlot(o, d).value ← newValue;
    Method do ⊥;
    Accessor do
      if not relaxedHasType(newValue, d.type) then throw typeError end if;
      d.f.call(o, ArgumentList⟨[newValue], {}⟩)
  end case
end proc;

proc writeDynamicProperty(o: Prototype ∪ DynamicInstance, name: String, newValue: Object)
  if some p ∈ o.dynamicProperties satisfies p.name = name then p.value ← newValue
  else
    o.dynamicProperties ← o.dynamicProperties ∪ {new DynamicProperty⟨⟨name, newValue⟩⟩}
  end if
end proc;
```

10.5.3 Lookup

~~10.3.1Reading 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.~~

~~**proc** *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**~~
~~**end proc**~~

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

proc *mostSpecificMember*(*c*: CLASSOPT, *global*: BOOLEAN, *qn*: QUALIFIEDNAME, *accesses*: MEMBERACCESS{}, *indexableOnly*: BOOLEAN): MEMBERDATAOPT
if *c* = **null then return null end if**;
qn2: QUALIFIEDNAME ← *qn*;
members: MEMBER{} ← *global* ? *c*.globalMembers : *c*.instanceMembers;
if some *m* ∈ *members* **satisfies** *m*.access ∈ *accesses* **and** *qn*.name = *m*.name **and**
qn.namespace ∈ *m*.namespaces **and** (**not** *indexableOnly* **or** *m*.indexable) **then**
d: MEMBERDATA ∪ NAMESPACE ← *m*.data;
if *d* ∉ NAMESPACE **then return** *d* **end if**;
qn2 ← QUALIFIEDNAME⟨*d*, *qn*.name⟩
end if;
return *mostSpecificMember*(*c*.super, *global*, *qn2*, *accesses*, *indexableOnly*)
end proc;

proc *resolveMemberNamespace*(*c*: CLASS, *global*: BOOLEAN, *pn*: PARTIALNAME, *accesses*: MEMBERACCESS{}): NAMESPACEOPT
  s: CLASSOPT ← *c*.super;
  if *s* ≠ **null then**
    ns: NAMESPACEOPT ← *resolveMemberNamespace*(*s*, *global*, *pn*, *accesses*);
    if *ns* ≠ **null then return** *ns* **end if**
  end if;
  members: MEMBER{} ← *global* ? *c*.globalMembers : *c*.instanceMembers;
  matches: MEMBER{} ← {*m* | ∀*m* ∈ *members* **such that**
      m.access ∈ *accesses* **and** *pn*.name = *m*.name **and** *pn*.namespaces ∩ *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*.
    matchingNamespaces: NAMESPACE{} ← *pn*.namespaces ∩ *match*.namespaces;
    Let *ns2*: NAMESPACE be any element of *matchingNamespaces*.
    return *ns2*
  end if;
  return null
end proc;

resolveObjectNamespace(*o*, *pn*, *accesses*) finds a namespace to use when reading or writing an unqualified property by searching for a member in the *least* derived ancestor that matches the name and has one of the namespaces given in *pn*. If no member is found, *resolveObjectNamespace* returns the `public` namespace if `public` was one of the namespaces in *pn* or raises an error if not.

proc *resolveObjectNamespace*(*o*: OBJECT, *pn*: PARTIALNAME, *accesses*: MEMBERACCESS{}): NAMESPACE
  ns: NAMESPACEOPT ← *o* ∈ CLASS ? *resolveMemberNamespace*(*o*, **true**, *pn*, *accesses*) :
      resolveMemberNamespace(*objectType*(*o*), **false**, *pn*, *accesses*);
  if *ns* ≠ **null then return** *ns* **end if**;
  if *publicNamespace* ∈ *pn*.namespaces **then return** *publicNamespace* **end if**;
  throw propertyNotFoundError
end proc;

~~**proc** *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**~~

~~**end proc**~~

~~10.3.2Reading 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 `used` around the point of the reference.~~

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

~~**proc** *readUnqualifiedProperty*(*o*: OBJECT, *name*: STRING, *uses*: NAMESPACE{}): OBJECT~~

~~*ns*: NAMESPACE ← *resolveObjectNamespace*(*o*, *name*, *uses*);~~

~~**return** *readQualifiedProperty*(*o*, *name*, *ns*, **false**)~~

~~**end proc**~~

~~*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.~~

~~**proc** *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*~~

~~**end proc**~~

~~*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**.~~

~~**proc** *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**~~
~~**end proc**~~

~~10.4~~10.6 Operator Dispatch

~~10.4.1~~10.6.1 Unary Operators

unaryDispatch(*table*, *limit*, *this*, ~~*op*~~*operand*, *args*) dispatches the unary operator described by *table* applied to the `this` value *this*, the ~~first argument~~operand ~~*op*~~*operand*, and zero or more ~~additional~~ positional and/or named arguments *args*. If *operand* has a non-**null** limit class, lookup is restricted to operators defined on the proper ancestors of that limit.~~If *limit* is non-**null**, lookup is restricted to operators defined on the proper superclasses of *limit*.~~

proc *unaryDispatch*(*table*: UNARYMETHOD{}, *this*: OBJECT, *operand*: OBJOPTIONALLIMIT, *args*: ARGUMENTLIST): OBJECT
applicableOps: UNARYMETHOD{} ← {*m* | ∀*m* ∈ *table* **such that** *limitedHasType*(*operand*, *m*.operandType)};
if there is some *best* ∈ *applicableOps* such that, given the choice of *best*, for every *m2* ∈ *applicableOps*, *m2*.operandType is an *ancestor* (see 9.1.8) of *best*.operandType **then**
return *best*.f(*this*, *getObject*(*operand*), *args*)
end if;
throw propertyNotFoundError
end proc;

limitedHasType(*o*, *c*) returns **true** if *o* is a member of class *c* with the added condition that, if *o* has a non-**null** limit class *limit*, *c* is a proper ancestor of *limit*.

proc *limitedHasType*(*o*: OBJOPTIONALLIMIT, *c*: CLASS): BOOLEAN
a: OBJECT ← *getObject*(*o*);
limit: CLASSOPT ← *getObjectLimit*(*o*);
if *hasType*(*a*, *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
end proc;

~~**proc** *unaryDispatch*(*table*: UNARYTABLE, *limit*: CLASSOPT, *this*: OBJECT, *op*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~Let *applicableMethods*: UNARYMETHOD{} be the set of all *m* ∈ *table*.methods such that~~
~~*limitedHasType*(*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 *best*: UNARYMETHOD be that element.~~
~~**return** *best*.op(*this*, *op*, *args*)~~
~~**end proc**~~

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

~~**proc** *limitedHasType*(*o*: OBJECT, *c*: CLASS, *limit*: CLASSOPT): BOOLEAN~~
~~**if** *hasType*(*o*, *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**~~
~~**end proc**~~

~~10.4.2~~10.6.2 Binary Operators

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*, *left*, *right*) dispatches the binary operator described by *table* applied to the operands *left* and *right*. If *left* has a non-**null** limit *leftLimit*, the lookup is restricted to operator definitions with an ancestor of *leftLimit* for the left operand. Similarly, if *right* has a non-**null** limit *rightLimit*, the lookup is restricted to operator definitions with an ancestor of *rightLimit* for the right operand.

proc *binaryDispatch*(*table*: BINARYMETHOD{}, *left*: OBJOPTIONALLIMIT, *right*: OBJOPTIONALLIMIT): OBJECT
applicableOps: BINARYMETHOD{} ← {*m* | ∀*m* ∈ *table* **such that**
limitedHasType(*left*, *m*.leftType) **and** *limitedHasType*(*right*, *m*.rightType)};
if there is some *best* ∈ *applicableOps* such that, given the choice of *best*, for every *m2* ∈ *applicableOps*, *best* is *at least as specific as m2* **then**
return *best*.f(*getObject*(*left*), *getObject*(*right*))
end if;
throw propertyNotFoundError
end proc;

~~*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.~~

~~**proc** *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~~
~~*limitedHasType*(*left*, *m*.leftType, *leftLimit*) = **true** and~~
~~*limitedHasType*(*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 *best*: BINARYMETHOD be that element.~~
~~**return** *best*.op(*left*, *right*)~~
~~**end proc**~~

~~10.5~~10.7 Name Lookup

11 Evaluation

11.1 Phases of Evaluation

- Parse using the grammar. If the parse fails, throw a syntax error.
- Call *Validate* on the goal nonterminal, which will recursively call *Validate* on some intermediate nonterminals. This checks that the program is well-formed, ensuring for instance that `break` and `continue` labels exist, compile-time

constant expressions really are compile-time constant expressions, etc. If the check fails, *Validate* will throw an exception.

- Call *Eval* on the goal nonterminal.

11.2 Constant Expressions

12 Expressions

Some expression grammar productions in this chapter are parameterised (see section 5.14.4) by the grammar argument β:

~~**Syntax**~~

$\beta \in$ {allowIn, noIn}

Most expression productions have both the *Validate* and *Eval* actions defined. Most of the *Eval* actions on subexpressions produce an OBJORREF result, indicating that the subexpression may evaluate to either a value or a place that can potentially be read, written, or deleted (see section 9.4).

12.1 Identifiers

An *Identifier* is either a non-keyword **Identifier** token or one of the non-reserved keywords `get`, `set`, `exclude`, `include`, or `named`. In either case, the *Name* action on the *Identifier* returns a string comprised of the identifier's characters after the lexer has processed any escape sequences.

Syntax

Identifier ⇒
  Identifier
| **`get`**
| **`set`**
| **`exclude`**
| **`include`**
| **`named`**

Semantics

Name[*Identifier*]: STRING;
  Name[*Identifier* ⇒ **Identifier**] = the string from the lexer's **identifier** token (see section 7)~~*Name*[**Identifier**]~~;
  Name[*Identifier* ⇒ **`get`**] = "`get`";
  Name[*Identifier* ⇒ **`set`**] = "`set`";
  Name[*Identifier* ⇒ **`exclude`**] = "`exclude`";
  Name[*Identifier* ⇒ **`include`**] = "`include`";
  Name[*Identifier* ⇒ **`named`**] = "`named`";

12.2 Qualified Identifiers

Syntax

Qualifier ⇒
  Identifier
| **`public`**
| **`private`**

SimpleQualifiedIdentifier ⇒
  Identifier
| *Qualifier* **`::`** *Identifier*

ExpressionQualifiedIdentifier ⇒ *ParenExpression* `::` *Identifier*

QualifiedIdentifier ⇒
  SimpleQualifiedIdentifier
| *ExpressionQualifiedIdentifier*

Validation

```
proc Validate[Qualifier] (v: VALIDATIONENV)
   [Qualifier ⇒ Identifier] do ????;
   [Qualifier ⇒ public] do nothing;
   [Qualifier ⇒ private] do if not insideClass(v) then throw syntaxError end if
end proc;

proc Validate[SimpleQualifiedIdentifier] (v: VALIDATIONENV)
   [SimpleQualifiedIdentifier ⇒ Identifier] do nothing;
   [SimpleQualifiedIdentifier ⇒ Qualifier :: Identifier] do Validate[Qualifier](v)
end proc;

proc Validate[ExpressionQualifiedIdentifier ⇒ ParenExpression :: Identifier] (v: VALIDATIONENV)
   Validate[ParenExpression](v);
   ????
end proc;

Validate[QualifiedIdentifier]: VALIDATIONENV → ();
   Validate[QualifiedIdentifier ⇒ SimpleQualifiedIdentifier] = Validate[SimpleQualifiedIdentifier];
   Validate[QualifiedIdentifier ⇒ ExpressionQualifiedIdentifier] = Validate[ExpressionQualifiedIdentifier];
```

Evaluation

```
proc Eval[Qualifier] (e: DYNAMICENV): NAMESPACE
   [Qualifier ⇒ Identifier] do
      a: OBJECT ← readReference(lookupVariable(e, Name[Identifier], true));
      if a ∉ NAMESPACE then throw typeError end if;
      return a;
   [Qualifier ⇒ public] do return publicNamespace;
   [Qualifier ⇒ private] do
      q: CLASSOPT ← e.enclosingClass;
      if q = null then ⊥ end if;
      return q.privateNamespace
end proc;

proc Eval[SimpleQualifiedIdentifier] (e: DYNAMICENV): OBJORREF
   [SimpleQualifiedIdentifier ⇒ Identifier] do
      return lookupVariable(e, Name[Identifier], false);
   [SimpleQualifiedIdentifier ⇒ Qualifier :: Identifier] do
      q: NAMESPACE ← Eval[Qualifier](e);
      return lookupQualifiedVariable(e, q, Name[Identifier])
end proc;

proc Eval[ExpressionQualifiedIdentifier ⇒ ParenExpression :: Identifier] (e: DYNAMICENV): OBJORREF
   q: OBJECT ← readReference(Eval[ParenExpression](e));
   if q ∉ NAMESPACE then throw typeError end if;
   return lookupQualifiedVariable(e, q, Name[Identifier])
end proc;
```

Eval[*QualifiedIdentifier*]: DYNAMICENV → OBJORREF;
  Eval[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] = *Eval*[*SimpleQualifiedIdentifier*];
  Eval[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] = *Eval*[*ExpressionQualifiedIdentifier*];

proc *Name*[*SimpleQualifiedIdentifier*] (*e*: DYNAMICENV): PARTIALNAME
  [*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do**
    return PARTIALNAME⟨*dynamicEnvUses*(*e*), *Name*[*Identifier*]⟩;
  [*SimpleQualifiedIdentifier* ⇒ *Qualifier* **::** *Identifier*] **do**
    q: NAMESPACE ← *Eval*[*Qualifier*](*e*);
    return PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

proc *Name*[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **::** *Identifier*] (*e*: DYNAMICENV): PARTIALNAME
  q: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
  if *q* ∉ NAMESPACE **then throw typeError end if**;
  return PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

Name[*QualifiedIdentifier*]: DYNAMICENV → PARTIALNAME;
  Name[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] = *Name*[*SimpleQualifiedIdentifier*];
  Name[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] = *Name*[*ExpressionQualifiedIdentifier*];

12.3 Unit Expressions

Syntax

UnitExpression ⇒
    ParenListExpression
  | **Number** [no line break] **String**
  | *UnitExpression* [no line break] **String**

Validation

proc *Validate*[*UnitExpression*] (*v*: VALIDATIONENV)
  [*UnitExpression* ⇒ *ParenListExpression*] **do** *Validate*[*ParenListExpression*](*v*);
  [*UnitExpression* ⇒ **Number** [no line break] **String**] **do** ????;
  [*UnitExpression* ⇒ *UnitExpression* [no line break] **String**] **do** ????
end proc;

Evaluation

proc *Eval*[*UnitExpression*] (*e*: DYNAMICENV): OBJORREF
  [*UnitExpression* ⇒ *ParenListExpression*] **do return** *Eval*[*ParenListExpression*](*e*);
  [*UnitExpression* ⇒ **Number** [no line break] **String**] **do** ????;
  [*UnitExpression* ⇒ *UnitExpression* [no line break] **String**] **do** ????
end proc;

12.4 Primary Expressions

Syntax

PrimaryExpression ⇒
  `null`
| **`true`**
| **`false`**
| **`public`**
| **Number**
| **String**
| **`this`**
| **RegularExpression**
| *UnitExpression*
| *ArrayLiteral*
| *ObjectLiteral*
| *FunctionExpression*

ParenExpression ⇒ **`(`** *AssignmentExpression*$^{\text{allowIn}}$ **`)`**

ParenListExpression ⇒
  ParenExpression
| **`(`** *ListExpression*$^{\text{allowIn}}$ **`,`** *AssignmentExpression*$^{\text{allowIn}}$ **`)`**

Validation

proc *Validate*[*PrimaryExpression*] (*v*: VALIDATIONENV)
  [*PrimaryExpression* ⇒ **`null`**] **do nothing**;
  [*PrimaryExpression* ⇒ **`true`**] **do nothing**;
  [*PrimaryExpression* ⇒ **`false`**] **do nothing**;
  [*PrimaryExpression* ⇒ **`public`**] **do nothing**;
  [*PrimaryExpression* ⇒ **Number**] **do nothing**;
  [*PrimaryExpression* ⇒ **String**] **do nothing**;
  [*PrimaryExpression* ⇒ **`this`**] **do** ????;
  [*PrimaryExpression* ⇒ **RegularExpression**] **do nothing**;
  [*PrimaryExpression* ⇒ *UnitExpression*] **do** *Validate*[*UnitExpression*](*v*);
  [*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *ObjectLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *FunctionExpression*] **do** *Validate*[*FunctionExpression*](*v*)
end proc;

Validate[*ParenExpression* ⇒ **`(`** *AssignmentExpression*$^{\text{allowIn}}$ **`)`**]: VALIDATIONENV → ()
    = *Validate*[*AssignmentExpression*$^{\text{allowIn}}$];

proc *Validate*[*ParenListExpression*] (*v*: VALIDATIONENV)
  [*ParenListExpression* ⇒ *ParenExpression*] **do** *Validate*[*ParenExpression*](*v*);
  [*ParenListExpression* ⇒ **`(`** *ListExpression*$^{\text{allowIn}}$ **`,`** *AssignmentExpression*$^{\text{allowIn}}$ **`)`**] **do**
    Validate[*ListExpression*$^{\text{allowIn}}$](*v*);
    Validate[*AssignmentExpression*$^{\text{allowIn}}$](*v*)
end proc;

Evaluation

proc *Eval*[*PrimaryExpression*] (*e*: DYNAMICENV): OBJORREF
  [*PrimaryExpression* ⇒ `null`] **do return null**;
  [*PrimaryExpression* ⇒ `true`] **do return true**;
  [*PrimaryExpression* ⇒ `false`] **do return false**;
  [*PrimaryExpression* ⇒ `public`] **do return** *publicNamespace*;
  [*PrimaryExpression* ⇒ **Number**] **do return** *Eval*[**Number**];
  [*PrimaryExpression* ⇒ **String**] **do return** *Eval*[**String**];
  [*PrimaryExpression* ⇒ `this`] **do return** *lookupThis*(*e*);
  [*PrimaryExpression* ⇒ **RegularExpression**] **do** ????;
  [*PrimaryExpression* ⇒ *UnitExpression*] **do return** *Eval*[*UnitExpression*](*e*);
  [*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *ObjectLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *FunctionExpression*] **do return** *Eval*[*FunctionExpression*](*e*)
end proc;

Eval[*ParenExpression* ⇒ `(` *AssignmentExpression*$^{\text{allowIn}}$ `)`]: DYNAMICENV → OBJORREF
  = *Eval*[*AssignmentExpression*$^{\text{allowIn}}$];

proc *Eval*[*ParenListExpression*] (*e*: DYNAMICENV): OBJORREF
  [*ParenListExpression* ⇒ *ParenExpression*] **do return** *Eval*[*ParenExpression*](*e*);
  [*ParenListExpression* ⇒ `(` *ListExpression*$^{\text{allowIn}}$ `,` *AssignmentExpression*$^{\text{allowIn}}$ `)`] **do**
    readReference(*Eval*[*ListExpression*$^{\text{allowIn}}$](*e*));
    return *readReference*(*Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*))
    ~~**return** *Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*)~~
end proc;

proc *EvalAsList*[*ParenListExpression*] (*e*: DYNAMICENV): OBJECT[]
  [*ParenListExpression* ⇒ *ParenExpression*] **do**
    elt: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
    return **[***elt***]**;
  [*ParenListExpression* ⇒ `(` *ListExpression*$^{\text{allowIn}}$ `,` *AssignmentExpression*$^{\text{allowIn}}$ `)`] **do**
    elts: OBJECT[] ← *EvalAsList*[*ListExpression*$^{\text{allowIn}}$](*e*);
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*));
    return *elts* ⊕ **[***elt***]**
end proc;

12.5 Function Expressions

Syntax

FunctionExpression ⇒
  `function` *FunctionSignature Block*
  | `function` *Identifier FunctionSignature Block*

Validation

proc *Validate*[*FunctionExpression*] (*v*: VALIDATIONENV)
  [*FunctionExpression* ⇒ `function` *FunctionSignature Block*] **do** ????;
  [*FunctionExpression* ⇒ `function` *Identifier FunctionSignature Block*] **do** ????
end proc;

Evaluation

proc *Eval*[*FunctionExpression*] (*e*: DYNAMICENV): OBJORREF
[*FunctionExpression* ⇒ **function** *FunctionSignature Block*] **do** ????;
[*FunctionExpression* ⇒ **function** *Identifier FunctionSignature Block*] **do** ????
end proc;

12.6 Object Literals

Syntax

ObjectLiteral ⇒
{ }
| **{** *FieldList* **}**

FieldList ⇒
LiteralField
| *FieldList* **,** *LiteralField*

LiteralField ⇒ *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$

FieldName ⇒
Identifier
| **String**
| **Number**
| *ParenExpression*

Validation

proc *Validate*[*LiteralField* ⇒ *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$] (*v*: VALIDATIONENV): STRING{}
names: STRING{} ← *Validate*[*FieldName*](*v*);
Validate[*AssignmentExpression*$^{\text{allowIn}}$](*v*);
return *names*
end proc;

proc *Validate*[*FieldName*] (*v*: VALIDATIONENV): STRING{}
[*FieldName* ⇒ *Identifier*] **do return** {*Name*[*Identifier*]};
[*FieldName* ⇒ **String**] **do return** {*Eval*[**String**]};
[*FieldName* ⇒ **Number**] **do** ????;
[*FieldName* ⇒ *ParenExpression*] **do** ????
end proc;

Evaluation

proc *Eval*[*LiteralField* ⇒ *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$] (*e*: DYNAMICENV): NAMEDARGUMENT
name: STRING ← *Eval*[*FieldName*](*e*);
value: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*));
return NAMEDARGUMENT⟨*name*, *value*⟩
end proc;

proc *Eval*[*FieldName*] (*e*: DYNAMICENV): STRING
[*FieldName* ⇒ *Identifier*] **do return** *Name*[*Identifier*];
[*FieldName* ⇒ **String**] **do return** *Eval*[**String**];
[*FieldName* ⇒ **Number**] **do** ????;
[*FieldName* ⇒ *ParenExpression*] **do** ????
end proc;

12.7 Array Literals

Syntax

ArrayLiteral ⇒ **`[`** *ElementList* **`]`**

ElementList ⇒
  LiteralElement
| *ElementList* **`,`** *LiteralElement*

LiteralElement ⇒
  «empty»
| *AssignmentExpression*^allowIn

12.8 Super Expressions

Syntax

SuperExpression ⇒
  `super`
| *FullSuperExpression*

FullSuperExpression ⇒ **`super`** *ParenExpression*

Validation

proc *Validate*[*SuperExpression*] (*v*: VALIDATIONENV)
  [*SuperExpression* ⇒ **`super`**] **do if not** *insideClass*(*v*) **then throw syntaxError end if**;
  [*SuperExpression* ⇒ *FullSuperExpression*] **do** *Validate*[*FullSuperExpression*](*v*)
end proc;

proc *Validate*[*FullSuperExpression* ⇒ **`super`** *ParenExpression*] (*v*: VALIDATIONENV)
  if not *insideClass*(*v*) **then throw syntaxError end if**;
  Validate[*ParenExpression*](*v*)
end proc;

Evaluation

proc *Eval*[*SuperExpression*] (*e*: DYNAMICENV): OBJORREFOPTIONALLIMIT
  [*SuperExpression* ⇒ **`super`**] **do**
    this: OBJECT ← *lookupThis*(*e*);
    limit: CLASS ← *lexicalClass*(*e*);
    return LIMITEDOBJORREF⟨*this*, *limit*⟩;
  [*SuperExpression* ⇒ *FullSuperExpression*] **do return** *Eval*[*FullSuperExpression*](*e*)
end proc;

proc *Eval*[*FullSuperExpression* ⇒ **`super`** *ParenExpression*] (*e*: DYNAMICENV): OBJORREFOPTIONALLIMIT
  r: OBJORREF ← *Eval*[*ParenExpression*](*e*);
  limit: CLASS ← *lexicalClass*(*e*);
  return LIMITEDOBJORREF⟨*r*, *limit*⟩
end proc;

12.9 Postfix Expressions

Syntax

PostfixExpression ⇒
  AttributeExpression
| *FullPostfixExpression*
| *ShortNewExpression*

PostfixExpressionOrSuper ⇒
  PostfixExpression
| *SuperExpression*

AttributeExpression ⇒
  SimpleQualifiedIdentifier
| *AttributeExpression MemberOperator*
| *AttributeExpression Arguments*

FullPostfixExpression ⇒
  PrimaryExpression
| *ExpressionQualifiedIdentifier*
| *FullNewExpression*
| *FullPostfixExpression MemberOperator*
| *SuperExpression DotOperator*
| *FullPostfixExpression Arguments*
| *FullSuperExpression Arguments*
| *PostfixExpressionOrSuper* [no line break] **++**
| *PostfixExpressionOrSuper* [no line break] **--**

FullNewExpression ⇒
  new *FullNewSubexpression Arguments*
| **new** *FullSuperExpression Arguments*

FullNewSubexpression ⇒
  PrimaryExpression
| *QualifiedIdentifier*
| *FullNewExpression*
| *FullNewSubexpression MemberOperator*
| *SuperExpression DotOperator*

ShortNewExpression ⇒
  new *ShortNewSubexpression*
| **new** *SuperExpression*

ShortNewSubexpression ⇒
  FullNewSubexpression
| *ShortNewExpression*

Validation

Validate[*PostfixExpression*]: VALIDATIONENV → ();
  Validate[*PostfixExpression* ⇒ *AttributeExpression*] = *Validate*[*AttributeExpression*];
  Validate[*PostfixExpression* ⇒ *FullPostfixExpression*] = *Validate*[*FullPostfixExpression*];
  Validate[*PostfixExpression* ⇒ *ShortNewExpression*] = *Validate*[*ShortNewExpression*];

Validate[*PostfixExpressionOrSuper*]: VALIDATIONENV → ();
  Validate[*PostfixExpressionOrSuper* ⇒ *PostfixExpression*] = *Validate*[*PostfixExpression*];
  Validate[*PostfixExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

proc *Validate*[*AttributeExpression*] (*v*: VALIDATIONENV)
  [*AttributeExpression* ⇒ *SimpleQualifiedIdentifier*] **do**
    Validate[*SimpleQualifiedIdentifier*](*v*);
  [$AttributeExpression_0$ ⇒ $AttributeExpression_1$ *MemberOperator*] **do**
    Validate[$AttributeExpression_1$](*v*);
    Validate[*MemberOperator*](*v*);
  [$AttributeExpression_0$ ⇒ $AttributeExpression_1$ *Arguments*] **do**
    Validate[$AttributeExpression_1$](*v*);
    Validate[*Arguments*](*v*)
end proc;

proc *Validate*[*FullPostfixExpression*] (*v*: VALIDATIONENV)
  [*FullPostfixExpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*v*);
  [*FullPostfixExpression* ⇒ *ExpressionQualifiedIdentifier*] **do**
    Validate[*ExpressionQualifiedIdentifier*](*v*);
  [*FullPostfixExpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*v*);
  [$FullPostfixExpression_0$ ⇒ $FullPostfixExpression_1$ *MemberOperator*] **do**
    Validate[$FullPostfixExpression_1$](*v*);
    Validate[*MemberOperator*](*v*);
  [*FullPostfixExpression* ⇒ *SuperExpression DotOperator*] **do**
    Validate[*SuperExpression*](*v*);
    Validate[*DotOperator*](*v*);
  [$FullPostfixExpression_0$ ⇒ $FullPostfixExpression_1$ *Arguments*] **do**
    Validate[$FullPostfixExpression_1$](*v*);
    Validate[*Arguments*](*v*);
  [*FullPostfixExpression* ⇒ *FullSuperExpression Arguments*] **do**
    Validate[*FullSuperExpression*](*v*);
    Validate[*Arguments*](*v*);
  [*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **++**] **do**
    Validate[*PostfixExpressionOrSuper*](*v*);
  [*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **--**] **do**
    Validate[*PostfixExpressionOrSuper*](*v*)
end proc;

proc *Validate*[*FullNewExpression*] (*v*: VALIDATIONENV)
  [*FullNewExpression* ⇒ **new** *FullNewSubexpression Arguments*] **do**
    Validate[*FullNewSubexpression*](*v*);
    Validate[*Arguments*](*v*);
  [*FullNewExpression* ⇒ **new** *FullSuperExpression Arguments*] **do**
    Validate[*FullSuperExpression*](*v*);
    Validate[*Arguments*](*v*)
end proc;

proc *Validate*[*FullNewSubexpression*] (*v*: VALIDATIONENV)
  [*FullNewSubexpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*v*);
  [*FullNewSubexpression* ⇒ *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*v*);
  [*FullNewSubexpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*v*);
  [$FullNewSubexpression_0$ ⇒ $FullNewSubexpression_1$ *MemberOperator*] **do**
    Validate[$FullNewSubexpression_1$](*v*);
    Validate[*MemberOperator*](*v*);
  [*FullNewSubexpression* ⇒ *SuperExpression DotOperator*] **do**
    Validate[*SuperExpression*](*v*);
    Validate[*DotOperator*](*v*)
end proc;

proc *Validate*[*ShortNewExpression*] (*v*: VALIDATIONENV)
  [*ShortNewExpression* ⇒ **new** *ShortNewSubexpression*] **do**
    Validate[*ShortNewSubexpression*](*v*);
  [*ShortNewExpression* ⇒ **new** *SuperExpression*] **do** *Validate*[*SuperExpression*](*v*)
end proc;

Validate[*ShortNewSubexpression*]: VALIDATIONENV → ();
  Validate[*ShortNewSubexpression* ⇒ *FullNewSubexpression*] = *Validate*[*FullNewSubexpression*];
  Validate[*ShortNewSubexpression* ⇒ *ShortNewExpression*] = *Validate*[*ShortNewExpression*];

Evaluation

Eval[*PostfixExpression*]: DYNAMICENV → OBJORREF;
  Eval[*PostfixExpression* ⇒ *AttributeExpression*] = *Eval*[*AttributeExpression*];
  Eval[*PostfixExpression* ⇒ *FullPostfixExpression*] = *Eval*[*FullPostfixExpression*];
  Eval[*PostfixExpression* ⇒ *ShortNewExpression*] = *Eval*[*ShortNewExpression*];

Eval[*PostfixExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*PostfixExpressionOrSuper* ⇒ *PostfixExpression*] = *Eval*[*PostfixExpression*];
  Eval[*PostfixExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

proc *Eval*[*AttributeExpression*] (*e*: DYNAMICENV): OBJORREF
  [*AttributeExpression* ⇒ *SimpleQualifiedIdentifier*] **do**
    return *Eval*[*SimpleQualifiedIdentifier*](*e*);
  [$AttributeExpression_0$ ⇒ $AttributeExpression_1$ *MemberOperator*] **do**
    a: OBJECT ← *readReference*(*Eval*[$AttributeExpression_1$](*e*));
    return *Eval*[*MemberOperator*](*e*, *a*);
  [$AttributeExpression_0$ ⇒ $AttributeExpression_1$ *Arguments*] **do**
    r: OBJORREF ← *Eval*[$AttributeExpression_1$](*e*);
    f: OBJECT ← *readReference*(*r*);
    base: OBJECT ← *referenceBase*(*r*);
    args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
    return *unaryDispatch*(*callTable*, *base*, *f*, *args*)
end proc;

proc *Eval*[*FullPostfixExpression*] (*e*: DYNAMICENV): OBJORREF
  [*FullPostfixExpression* ⇒ *PrimaryExpression*] **do return** *Eval*[*PrimaryExpression*](*e*);
  [*FullPostfixExpression* ⇒ *ExpressionQualifiedIdentifier*] **do**
    return *Eval*[*ExpressionQualifiedIdentifier*](*e*);
  [*FullPostfixExpression* ⇒ *FullNewExpression*] **do return** *Eval*[*FullNewExpression*](*e*);
  [$FullPostfixExpression_0$ ⇒ $FullPostfixExpression_1$ *MemberOperator*] **do**
    a: OBJECT ← *readReference*(*Eval*[$FullPostfixExpression_1$](*e*));
    return *Eval*[*MemberOperator*](*e*, *a*);
  [*FullPostfixExpression* ⇒ *SuperExpression DotOperator*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*SuperExpression*](*e*));
    return *Eval*[*DotOperator*](*e*, *a*);
  [$FullPostfixExpression_0$ ⇒ $FullPostfixExpression_1$ *Arguments*] **do**
    r: OBJORREF ← *Eval*[$FullPostfixExpression_1$](*e*);
    f: OBJECT ← *readReference*(*r*);
    base: OBJECT ← *referenceBase*(*r*);
    args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
    return *unaryDispatch*(*callTable*, *base*, *f*, *args*);

[*FullPostfixExpression* ⇒ *FullSuperExpression Arguments*] **do**
  r: OBJORREFOPTIONALLIMIT ← *Eval*[*FullSuperExpression*](*e*);
  f: OBJOPTIONALLIMIT ← *readRefWithLimit*(*r*);
  base: OBJECT ← *referenceBase*(*r*);
  args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
  return *unaryDispatch*(*callTable*, *base*, *f*, *args*);
[*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **++**] **do**
  r: OBJORREFOPTIONALLIMIT ← *Eval*[*PostfixExpressionOrSuper*](*e*);
  a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*r*);
  b: OBJECT ← *unaryDispatch*(*incrementTable*, **null**, *a*, ARGUMENTLIST⟨[], {}⟩);
  writeReference(*r*, *b*);
  return *getObject*(*a*);
[*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **--**] **do**
  r: OBJORREFOPTIONALLIMIT ← *Eval*[*PostfixExpressionOrSuper*](*e*);
  a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*r*);
  b: OBJECT ← *unaryDispatch*(*decrementTable*, **null**, *a*, ARGUMENTLIST⟨[], {}⟩);
  writeReference(*r*, *b*);
  return *getObject*(*a*)
end proc;

proc *Eval*[*FullNewExpression*] (*e*: DYNAMICENV): OBJORREF
  [*FullNewExpression* ⇒ **new** *FullNewSubexpression Arguments*] **do**
    f: OBJECT ← *readReference*(*Eval*[*FullNewSubexpression*](*e*));
    args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
    return *unaryDispatch*(*constructTable*, **null**, *f*, *args*);
  [*FullNewExpression* ⇒ **new** *FullSuperExpression Arguments*] **do**
    f: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*FullSuperExpression*](*e*));
    args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
    return *unaryDispatch*(*constructTable*, **null**, *f*, *args*)
end proc;

proc *Eval*[*FullNewSubexpression*] (*e*: DYNAMICENV): OBJORREF
  [*FullNewSubexpression* ⇒ *PrimaryExpression*] **do return** *Eval*[*PrimaryExpression*](*e*);
  [*FullNewSubexpression* ⇒ *QualifiedIdentifier*] **do return** *Eval*[*QualifiedIdentifier*](*e*);
  [*FullNewSubexpression* ⇒ *FullNewExpression*] **do return** *Eval*[*FullNewExpression*](*e*);
  [$FullNewSubexpression_0$ ⇒ $FullNewSubexpression_1$ *MemberOperator*] **do**
    a: OBJECT ← *readReference*(*Eval*[$FullNewSubexpression_1$](*e*));
    return *Eval*[*MemberOperator*](*e*, *a*);
  [*FullNewSubexpression* ⇒ *SuperExpression DotOperator*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*SuperExpression*](*e*));
    return *Eval*[*DotOperator*](*e*, *a*)
end proc;

proc *Eval*[*ShortNewExpression*] (*e*: DYNAMICENV): OBJORREF
  [*ShortNewExpression* ⇒ **new** *ShortNewSubexpression*] **do**
    f: OBJECT ← *readReference*(*Eval*[*ShortNewSubexpression*](*e*));
    return *unaryDispatch*(*constructTable*, **null**, *f*, ARGUMENTLIST⟨[], {}⟩);
  [*ShortNewExpression* ⇒ **new** *SuperExpression*] **do**
    f: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*SuperExpression*](*e*));
    return *unaryDispatch*(*constructTable*, **null**, *f*, ARGUMENTLIST⟨[], {}⟩)
end proc;

Eval[*ShortNewSubexpression*]: DYNAMICENV → OBJORREF;
  Eval[*ShortNewSubexpression* ⇒ *FullNewSubexpression*] = *Eval*[*FullNewSubexpression*];
  Eval[*ShortNewSubexpression* ⇒ *ShortNewExpression*] = *Eval*[*ShortNewExpression*];

12.10 Member Operators

Syntax

MemberOperator ⇒
  DotOperator
| **.** *ParenExpression*

DotOperator ⇒
  . *QualifiedIdentifier*
| *Brackets*

Brackets ⇒
  [ ]
| **[** *ListExpression*$^{\text{allowIn}}$ **]**
| **[** *NamedArgumentList* **]**

Arguments ⇒
  ParenExpressions
| **(** *NamedArgumentList* **)**

ParenExpressions ⇒
  ( )
| *ParenListExpression*

NamedArgumentList ⇒
  LiteralField
| *ListExpression*$^{\text{allowIn}}$ **,** *LiteralField*
| *NamedArgumentList* **,** *LiteralField*

Validation

proc *Validate*[*MemberOperator*] (*v*: VALIDATIONENV)
  [*MemberOperator* ⇒ *DotOperator*] **do** *Validate*[*DotOperator*](*v*);
  [*MemberOperator* ⇒ **.** *ParenExpression*] **do** *Validate*[*ParenExpression*](*v*)
end proc;

proc *Validate*[*DotOperator*] (*v*: VALIDATIONENV)
  [*DotOperator* ⇒ **.** *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*v*);
  [*DotOperator* ⇒ *Brackets*] **do** *Validate*[*Brackets*](*v*)
end proc;

proc *Validate*[*Brackets*] (*v*: VALIDATIONENV)
  [*Brackets* ⇒ **[ ]**] **do nothing**;
  [*Brackets* ⇒ **[** *ListExpression*$^{\text{allowIn}}$ **]**] **do** *Validate*[*ListExpression*$^{\text{allowIn}}$](*v*);
  [*Brackets* ⇒ **[** *NamedArgumentList* **]**] **do** *Validate*[*NamedArgumentList*](*v*)
end proc;

proc *Validate*[*Arguments*] (*v*: VALIDATIONENV)
  [*Arguments* ⇒ *ParenExpressions*] **do** *Validate*[*ParenExpressions*](*v*);
  [*Arguments* ⇒ **(** *NamedArgumentList* **)**] **do** *Validate*[*NamedArgumentList*](*v*)
end proc;

proc *Validate*[*ParenExpressions*] (*v*: VALIDATIONENV)
  [*ParenExpressions* ⇒ **( )**] **do nothing**;
  [*ParenExpressions* ⇒ *ParenListExpression*] **do** *Validate*[*ParenListExpression*](*v*)
end proc;

```
proc Validate[NamedArgumentList] (v: ValidationEnv): String{}
   [NamedArgumentList ⇒ LiteralField] do return Validate[LiteralField](v);
   [NamedArgumentList ⇒ ListExpression^allowIn , LiteralField] do
      Validate[ListExpression^allowIn](v);
      return Validate[LiteralField](v);
   [NamedArgumentList_0 ⇒ NamedArgumentList_1 , LiteralField] do
      names1: String{} ← Validate[NamedArgumentList_1](v);
      names2: String{} ← Validate[LiteralField](v);
      if names1 ∩ names2 ≠ {} then throw syntaxError end if;
      return names1 ∪ names2
end proc;
```

Evaluation

```
proc Eval[MemberOperator] (e: DynamicEnv, base: Object): ObjOrRef
   [MemberOperator ⇒ DotOperator] do return Eval[DotOperator](e, base);
   [MemberOperator ⇒ . ParenExpression] do ????
end proc;

proc Eval[DotOperator] (e: DynamicEnv, base: ObjOptionalLimit): ObjOrRef
   [DotOperator ⇒ . QualifiedIdentifier] do
      n: PartialName ← Name[QualifiedIdentifier](e);
      return DotReference⟨base, n⟩;
   [DotOperator ⇒ Brackets] do
      args: ArgumentList ← Eval[Brackets](e);
      return BracketReference⟨base, args⟩
end proc;

proc Eval[Brackets] (e: DynamicEnv): ArgumentList
   [Brackets ⇒ [ ]] do return ArgumentList⟨[], {}⟩;
   [Brackets ⇒ [ ListExpression^allowIn ]] do
      positional: Object[] ← EvalAsList[ListExpression^allowIn](e);
      return ArgumentList⟨positional, {}⟩;
   [Brackets ⇒ [ NamedArgumentList ]] do return Eval[NamedArgumentList](e)
end proc;

proc Eval[Arguments] (e: DynamicEnv): ArgumentList
   [Arguments ⇒ ParenExpressions] do return Eval[ParenExpressions](e);
   [Arguments ⇒ ( NamedArgumentList )] do return Eval[NamedArgumentList](e)
end proc;

proc Eval[ParenExpressions] (e: DynamicEnv): ArgumentList
   [ParenExpressions ⇒ ( )] do return ArgumentList⟨[], {}⟩;
   [ParenExpressions ⇒ ParenListExpression] do
      positional: Object[] ← EvalAsList[ParenListExpression](e);
      return ArgumentList⟨positional, {}⟩
end proc;

proc Eval[NamedArgumentList] (e: DynamicEnv): ArgumentList
   [NamedArgumentList ⇒ LiteralField] do
      na: NamedArgument ← Eval[LiteralField](e);
      return ArgumentList⟨[], {na}⟩;
```

```
    [NamedArgumentList ⇒ ListExpression^allowIn , LiteralField] do
      positional: OBJECT[] ← EvalAsList[ListExpression^allowIn](e);
      na: NAMEDARGUMENT ← Eval[LiteralField](e);
      return ARGUMENTLIST⟨positional, {na}⟩;
    [NamedArgumentList0 ⇒ NamedArgumentList1 , LiteralField] do
      args: ARGUMENTLIST ← Eval[NamedArgumentList1](e);
      na: NAMEDARGUMENT ← Eval[LiteralField](e);
      if some na2 ∈ args.named satisfies na2.name = na.name then
        throw argumentMismatchError
      end if;
      return ARGUMENTLIST⟨args.positional, args.named ∪ {na}⟩
  end proc;
```

12.11 Unary Operators

Syntax

```
UnaryExpression ⇒
    PostfixExpression
  | delete PostfixExpression
  | void UnaryExpression
  | typeof UnaryExpression
  | ++ PostfixExpressionOrSuper
  | -- PostfixExpressionOrSuper
  | + UnaryExpressionOrSuper
  | - UnaryExpressionOrSuper
  | ~ UnaryExpressionOrSuper
  | ! UnaryExpression

UnaryExpressionOrSuper ⇒
    UnaryExpression
  | SuperExpression
```

Validation

```
  proc Validate[UnaryExpression] (v: VALIDATIONENV)
    [UnaryExpression ⇒ PostfixExpression] do Validate[PostfixExpression](v);
    [UnaryExpression ⇒ delete PostfixExpression] do Validate[PostfixExpression](v);
    [UnaryExpression0 ⇒ void UnaryExpression1] do Validate[UnaryExpression1](v);
    [UnaryExpression0 ⇒ typeof UnaryExpression1] do Validate[UnaryExpression1](v);
    [UnaryExpression ⇒ ++ PostfixExpressionOrSuper] do
      Validate[PostfixExpressionOrSuper](v);
    [UnaryExpression ⇒ -- PostfixExpressionOrSuper] do
      Validate[PostfixExpressionOrSuper](v);
    [UnaryExpression ⇒ + UnaryExpressionOrSuper] do Validate[UnaryExpressionOrSuper](v);
    [UnaryExpression ⇒ - UnaryExpressionOrSuper] do Validate[UnaryExpressionOrSuper](v);
    [UnaryExpression ⇒ ~ UnaryExpressionOrSuper] do Validate[UnaryExpressionOrSuper](v);
    [UnaryExpression0 ⇒ ! UnaryExpression1] do Validate[UnaryExpression1](v)
  end proc;

  Validate[UnaryExpressionOrSuper]: VALIDATIONENV → ();
    Validate[UnaryExpressionOrSuper ⇒ UnaryExpression] = Validate[UnaryExpression];
    Validate[UnaryExpressionOrSuper ⇒ SuperExpression] = Validate[SuperExpression];
```

Evaluation

```
proc Eval[UnaryExpression] (e: DynamicEnv): ObjOrRef
   [UnaryExpression ⇒ PostfixExpression] do return Eval[PostfixExpression](e);
   [UnaryExpression ⇒ delete PostfixExpression] do
      return deleteReference(Eval[PostfixExpression](e));
   [UnaryExpression0 ⇒ void UnaryExpression1] do
      readReference(Eval[UnaryExpression1](e));
      return undefined;
   [UnaryExpression0 ⇒ typeof UnaryExpression1] do
      a: Object ← readReference(Eval[UnaryExpression1](e));
      case a of
         Undefined do return "undefined";
         Null ∪ Prototype do return "object";
         Boolean do return "boolean";
         Float64 do return "number";
         String do return "string";
         Namespace do return "namespace";
         Attribute do return "attribute";
         Class ∪ MethodClosure do return "function";
         Instance do return a.typeofString
      end case;
   [UnaryExpression ⇒ ++ PostfixExpressionOrSuper] do
      r: ObjOrRefOptionalLimit ← Eval[PostfixExpressionOrSuper](e);
      a: ObjOptionalLimit ← readRefWithLimit(r);
      b: Object ← unaryDispatch(incrementTable, null, a, ArgumentList⟨[], {}⟩);
      writeReference(r, b);
      return b;
   [UnaryExpression ⇒ -- PostfixExpressionOrSuper] do
      r: ObjOrRefOptionalLimit ← Eval[PostfixExpressionOrSuper](e);
      a: ObjOptionalLimit ← readRefWithLimit(r);
      b: Object ← unaryDispatch(decrementTable, null, a, ArgumentList⟨[], {}⟩);
      writeReference(r, b);
      return b;
   [UnaryExpression ⇒ + UnaryExpressionOrSuper] do
      a: ObjOptionalLimit ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
      return unaryPlus(a);
   [UnaryExpression ⇒ - UnaryExpressionOrSuper] do
      a: ObjOptionalLimit ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
      return unaryDispatch(minusTable, null, a, ArgumentList⟨[], {}⟩);
   [UnaryExpression ⇒ ~ UnaryExpressionOrSuper] do
      a: ObjOptionalLimit ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
      return unaryDispatch(bitwiseNotTable, null, a, ArgumentList⟨[], {}⟩);
   [UnaryExpression0 ⇒ ! UnaryExpression1] do
      a: Object ← readReference(Eval[UnaryExpression1](e));
      return unaryNot(a)
end proc;

Eval[UnaryExpressionOrSuper]: DynamicEnv → ObjOrRefOptionalLimit;
   Eval[UnaryExpressionOrSuper ⇒ UnaryExpression] = Eval[UnaryExpression];
   Eval[UnaryExpressionOrSuper ⇒ SuperExpression] = Eval[SuperExpression];
```

12.12 Multiplicative Operators

Syntax

MultiplicativeExpression ⇒
  UnaryExpression
| *MultiplicativeExpressionOrSuper* ***** *UnaryExpressionOrSuper*
| *MultiplicativeExpressionOrSuper* **/** *UnaryExpressionOrSuper*
| *MultiplicativeExpressionOrSuper* **%** *UnaryExpressionOrSuper*

MultiplicativeExpressionOrSuper ⇒
  MultiplicativeExpression
| *SuperExpression*

Validation

proc *Validate*[*MultiplicativeExpression*] (*v*: VALIDATIONENV)
  [*MultiplicativeExpression* ⇒ *UnaryExpression*] **do** *Validate*[*UnaryExpression*](*v*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* ***** *UnaryExpressionOrSuper*] **do**
    Validate[*MultiplicativeExpressionOrSuper*](*v*);
    Validate[*UnaryExpressionOrSuper*](*v*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **/** *UnaryExpressionOrSuper*] **do**
    Validate[*MultiplicativeExpressionOrSuper*](*v*);
    Validate[*UnaryExpressionOrSuper*](*v*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **%** *UnaryExpressionOrSuper*] **do**
    Validate[*MultiplicativeExpressionOrSuper*](*v*);
    Validate[*UnaryExpressionOrSuper*](*v*)
end proc;

Validate[*MultiplicativeExpressionOrSuper*]: VALIDATIONENV → ();
  Validate[*MultiplicativeExpressionOrSuper* ⇒ *MultiplicativeExpression*] = *Validate*[*MultiplicativeExpression*];
  Validate[*MultiplicativeExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*MultiplicativeExpression*] (*e*: DYNAMICENV): OBJORREF
  [*MultiplicativeExpression* ⇒ *UnaryExpression*] **do return** *Eval*[*UnaryExpression*](*e*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* ***** *UnaryExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
    return *binaryDispatch*(*multiplyTable*, *a*, *b*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **/** *UnaryExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
    return *binaryDispatch*(*divideTable*, *a*, *b*);
  [*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **%** *UnaryExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
    return *binaryDispatch*(*remainderTable*, *a*, *b*)
end proc;

Eval[*MultiplicativeExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*MultiplicativeExpressionOrSuper* ⇒ *MultiplicativeExpression*] = *Eval*[*MultiplicativeExpression*];
  Eval[*MultiplicativeExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.13 Additive Operators

Syntax

AdditiveExpression ⇒
  MultiplicativeExpression
| *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*
| *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*

AdditiveExpressionOrSuper ⇒
  AdditiveExpression
| *SuperExpression*

Validation

proc *Validate*[*AdditiveExpression*] (*v*: VALIDATIONENV)
  [*AdditiveExpression* ⇒ *MultiplicativeExpression*] **do**
    Validate[*MultiplicativeExpression*](*v*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*] **do**
    Validate[*AdditiveExpressionOrSuper*](*v*);
    Validate[*MultiplicativeExpressionOrSuper*](*v*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*] **do**
    Validate[*AdditiveExpressionOrSuper*](*v*);
    Validate[*MultiplicativeExpressionOrSuper*](*v*)
end proc;

Validate[*AdditiveExpressionOrSuper*]: VALIDATIONENV → ();
  Validate[*AdditiveExpressionOrSuper* ⇒ *AdditiveExpression*] = *Validate*[*AdditiveExpression*];
  Validate[*AdditiveExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*AdditiveExpression*] (*e*: DYNAMICENV): OBJORREF
  [*AdditiveExpression* ⇒ *MultiplicativeExpression*] **do**
    return *Eval*[*MultiplicativeExpression*](*e*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    return *binaryDispatch*(*addTable*, *a*, *b*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    return *binaryDispatch*(*subtractTable*, *a*, *b*)
end proc;

Eval[*AdditiveExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*AdditiveExpressionOrSuper* ⇒ *AdditiveExpression*] = *Eval*[*AdditiveExpression*];
  Eval[*AdditiveExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.14 Bitwise Shift Operators

Syntax

ShiftExpression ⇒
  AdditiveExpression
| *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*
| *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*
| *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*

ShiftExpressionOrSuper ⇒
  ShiftExpression
| *SuperExpression*

Validation

proc *Validate*[*ShiftExpression*] (*v*: VALIDATIONENV)
  [*ShiftExpression* ⇒ *AdditiveExpression*] **do** *Validate*[*AdditiveExpression*](*v*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*] **do**
    Validate[*ShiftExpressionOrSuper*](*v*);
    Validate[*AdditiveExpressionOrSuper*](*v*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*] **do**
    Validate[*ShiftExpressionOrSuper*](*v*);
    Validate[*AdditiveExpressionOrSuper*](*v*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*] **do**
    Validate[*ShiftExpressionOrSuper*](*v*);
    Validate[*AdditiveExpressionOrSuper*](*v*)
end proc;

Validate[*ShiftExpressionOrSuper*]: VALIDATIONENV → ();
  Validate[*ShiftExpressionOrSuper* ⇒ *ShiftExpression*] = *Validate*[*ShiftExpression*];
  Validate[*ShiftExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*ShiftExpression*] (*e*: DYNAMICENV): OBJORREF
  [*ShiftExpression* ⇒ *AdditiveExpression*] **do return** *Eval*[*AdditiveExpression*](*e*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    return *binaryDispatch*(*shiftLeftTable*, *a*, *b*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    return *binaryDispatch*(*shiftRightTable*, *a*, *b*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    return *binaryDispatch*(*shiftRightUnsignedTable*, *a*, *b*)
end proc;

Eval[*ShiftExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*ShiftExpressionOrSuper* ⇒ *ShiftExpression*] = *Eval*[*ShiftExpression*];
  Eval[*ShiftExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.15 Relational Operators

Syntax

RelationalExpression$^{\text{allowIn}}$ ⇒
ShiftExpression
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **<** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **>** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **<=** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **>=** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{allowIn}}$ **is** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **as** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **in** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{allowIn}}$ **instanceof** *ShiftExpression*

RelationalExpression$^{\text{noIn}}$ ⇒
ShiftExpression
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **<** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **>** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **<=** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **>=** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{noIn}}$ **is** *ShiftExpression*
| *RelationalExpression*$^{\text{noIn}}$ **as** *ShiftExpression*
| *RelationalExpression*$^{\text{noIn}}$ **instanceof** *ShiftExpression*

RelationalExpressionOrSuper$^{\beta}$ ⇒
RelationalExpression$^{\beta}$
| *SuperExpression*

Validation

proc *Validate*[*RelationalExpression*$^{\beta}$] (*v*: VALIDATIONENV)
[*RelationalExpression*$^{\beta}$ ⇒ *ShiftExpression*] **do** *Validate*[*ShiftExpression*](*v*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<** *ShiftExpressionOrSuper*] **do**
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*v*);
Validate[*ShiftExpressionOrSuper*](*v*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>** *ShiftExpressionOrSuper*] **do**
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*v*);
Validate[*ShiftExpressionOrSuper*](*v*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<=** *ShiftExpressionOrSuper*] **do**
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*v*);
Validate[*ShiftExpressionOrSuper*](*v*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>=** *ShiftExpressionOrSuper*] **do**
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*v*);
Validate[*ShiftExpressionOrSuper*](*v*);
[*RelationalExpression*$^{\beta}_{0}$ ⇒ *RelationalExpression*$^{\beta}_{1}$ **is** *ShiftExpression*] **do**
Validate[*RelationalExpression*$^{\beta}_{1}$](*v*);
Validate[*ShiftExpression*](*v*);
[*RelationalExpression*$^{\beta}_{0}$ ⇒ *RelationalExpression*$^{\beta}_{1}$ **as** *ShiftExpression*] **do**
Validate[*RelationalExpression*$^{\beta}_{1}$](*v*);
Validate[*ShiftExpression*](*v*);
[*RelationalExpression*$^{\text{allowIn}}_{0}$ ⇒ *RelationalExpression*$^{\text{allowIn}}_{1}$ **in** *ShiftExpressionOrSuper*] **do**
Validate[*RelationalExpression*$^{\text{allowIn}}_{1}$](*v*);
Validate[*ShiftExpressionOrSuper*](*v*);

[*RelationalExpression*$^\beta{}_0$ ⇒ *RelationalExpression*$^\beta{}_1$ **instanceof** *ShiftExpression*] **do**
Validate[*RelationalExpression*$^\beta{}_1$](*v*);
Validate[*ShiftExpression*](*v*)
end proc;

Validate[*RelationalExpressionOrSuper*$^\beta$]: VALIDATIONENV → ();
Validate[*RelationalExpressionOrSuper*$^\beta$ ⇒ *RelationalExpression*$^\beta$] = *Validate*[*RelationalExpression*$^\beta$];
Validate[*RelationalExpressionOrSuper*$^\beta$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*RelationalExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
[*RelationalExpression*$^\beta$ ⇒ *ShiftExpression*] **do return** *Eval*[*ShiftExpression*](*e*);
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpressionOrSuper*$^\beta$ **<** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessTable*, *a*, *b*);
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpressionOrSuper*$^\beta$ **>** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessTable*, *b*, *a*);
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpressionOrSuper*$^\beta$ **<=** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessOrEqualTable*, *a*, *b*);
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpressionOrSuper*$^\beta$ **>=** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessOrEqualTable*, *b*, *a*);
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpression*$^\beta$ **is** *ShiftExpression*] **do** ????;
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpression*$^\beta$ **as** *ShiftExpression*] **do** ????;
[*RelationalExpression*$^{\text{allowIn}}$ ⇒ *RelationalExpression*$^{\text{allowIn}}$ **in** *ShiftExpressionOrSuper*] **do**
????;
[*RelationalExpression*$^\beta$ ⇒ *RelationalExpression*$^\beta$ **instanceof** *ShiftExpression*] **do** ????
end proc;

Eval[*RelationalExpressionOrSuper*$^\beta$]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
Eval[*RelationalExpressionOrSuper*$^\beta$ ⇒ *RelationalExpression*$^\beta$] = *Eval*[*RelationalExpression*$^\beta$];
Eval[*RelationalExpressionOrSuper*$^\beta$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.16 Equality Operators

Syntax

EqualityExpression$^\beta$ ⇒
RelationalExpression$^\beta$
| *EqualityExpressionOrSuper*$^\beta$ **==** *RelationalExpressionOrSuper*$^\beta$
| *EqualityExpressionOrSuper*$^\beta$ **!=** *RelationalExpressionOrSuper*$^\beta$
| *EqualityExpressionOrSuper*$^\beta$ **===** *RelationalExpressionOrSuper*$^\beta$
| *EqualityExpressionOrSuper*$^\beta$ **!==** *RelationalExpressionOrSuper*$^\beta$

EqualityExpressionOrSuper$^\beta$ ⇒
EqualityExpression$^\beta$
| *SuperExpression*

Validation

proc *Validate*[*EqualityExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*EqualityExpression*$^\beta$ ⇒ *RelationalExpression*$^\beta$] **do** *Validate*[*RelationalExpression*$^\beta$](*v*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **==** *RelationalExpressionOrSuper*$^\beta$] **do**
    Validate[*EqualityExpressionOrSuper*$^\beta$](*v*);
    Validate[*RelationalExpressionOrSuper*$^\beta$](*v*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **!=** *RelationalExpressionOrSuper*$^\beta$] **do**
    Validate[*EqualityExpressionOrSuper*$^\beta$](*v*);
    Validate[*RelationalExpressionOrSuper*$^\beta$](*v*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **===** *RelationalExpressionOrSuper*$^\beta$] **do**
    Validate[*EqualityExpressionOrSuper*$^\beta$](*v*);
    Validate[*RelationalExpressionOrSuper*$^\beta$](*v*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **!==** *RelationalExpressionOrSuper*$^\beta$] **do**
    Validate[*EqualityExpressionOrSuper*$^\beta$](*v*);
    Validate[*RelationalExpressionOrSuper*$^\beta$](*v*)
end proc;

Validate[*EqualityExpressionOrSuper*$^\beta$]: VALIDATIONENV → ();
  Validate[*EqualityExpressionOrSuper*$^\beta$ ⇒ *EqualityExpression*$^\beta$] = *Validate*[*EqualityExpression*$^\beta$];
  Validate[*EqualityExpressionOrSuper*$^\beta$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*EqualityExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*EqualityExpression*$^\beta$ ⇒ *RelationalExpression*$^\beta$] **do**
    return *Eval*[*RelationalExpression*$^\beta$](*e*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **==** *RelationalExpressionOrSuper*$^\beta$] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^\beta$](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
    return *binaryDispatch*(*equalTable*, *a*, *b*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **!=** *RelationalExpressionOrSuper*$^\beta$] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^\beta$](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
    return *unaryNot*(*binaryDispatch*(*equalTable*, *a*, *b*));
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **===** *RelationalExpressionOrSuper*$^\beta$] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^\beta$](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
    return *binaryDispatch*(*strictEqualTable*, *a*, *b*);
  [*EqualityExpression*$^\beta$ ⇒ *EqualityExpressionOrSuper*$^\beta$ **!==** *RelationalExpressionOrSuper*$^\beta$] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^\beta$](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^\beta$](*e*));
    return *unaryNot*(*binaryDispatch*(*strictEqualTable*, *a*, *b*))
end proc;

Eval[*EqualityExpressionOrSuper*$^\beta$]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*EqualityExpressionOrSuper*$^\beta$ ⇒ *EqualityExpression*$^\beta$] = *Eval*[*EqualityExpression*$^\beta$];
  Eval[*EqualityExpressionOrSuper*$^\beta$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.17 Binary Bitwise Operators

Syntax

BitwiseAndExpression$^\beta$ ⇒
  EqualityExpression$^\beta$
| *BitwiseAndExpressionOrSuper*$^\beta$ **&** *EqualityExpressionOrSuper*$^\beta$

BitwiseXorExpression$^\beta$ ⇒
  BitwiseAndExpression$^\beta$
| *BitwiseXorExpressionOrSuper*$^\beta$ **^** *BitwiseAndExpressionOrSuper*$^\beta$

BitwiseOrExpression$^\beta$ ⇒
  BitwiseXorExpression$^\beta$
| *BitwiseOrExpressionOrSuper*$^\beta$ **|** *BitwiseXorExpressionOrSuper*$^\beta$

BitwiseAndExpressionOrSuper$^\beta$ ⇒
  BitwiseAndExpression$^\beta$
| *SuperExpression*

BitwiseXorExpressionOrSuper$^\beta$ ⇒
  BitwiseXorExpression$^\beta$
| *SuperExpression*

BitwiseOrExpressionOrSuper$^\beta$ ⇒
  BitwiseOrExpression$^\beta$
| *SuperExpression*

Validation

proc *Validate*[*BitwiseAndExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*BitwiseAndExpression*$^\beta$ ⇒ *EqualityExpression*$^\beta$] **do** *Validate*[*EqualityExpression*$^\beta$](*v*);
  [*BitwiseAndExpression*$^\beta$ ⇒ *BitwiseAndExpressionOrSuper*$^\beta$ **&** *EqualityExpressionOrSuper*$^\beta$] **do**
    Validate[*BitwiseAndExpressionOrSuper*$^\beta$](*v*);
    Validate[*EqualityExpressionOrSuper*$^\beta$](*v*)
end proc;

proc *Validate*[*BitwiseXorExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*BitwiseXorExpression*$^\beta$ ⇒ *BitwiseAndExpression*$^\beta$] **do**
    Validate[*BitwiseAndExpression*$^\beta$](*v*);
  [*BitwiseXorExpression*$^\beta$ ⇒ *BitwiseXorExpressionOrSuper*$^\beta$ **^** *BitwiseAndExpressionOrSuper*$^\beta$] **do**
    Validate[*BitwiseXorExpressionOrSuper*$^\beta$](*v*);
    Validate[*BitwiseAndExpressionOrSuper*$^\beta$](*v*)
end proc;

proc *Validate*[*BitwiseOrExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*BitwiseOrExpression*$^\beta$ ⇒ *BitwiseXorExpression*$^\beta$] **do**
    Validate[*BitwiseXorExpression*$^\beta$](*v*);
  [*BitwiseOrExpression*$^\beta$ ⇒ *BitwiseOrExpressionOrSuper*$^\beta$ **|** *BitwiseXorExpressionOrSuper*$^\beta$] **do**
    Validate[*BitwiseOrExpressionOrSuper*$^\beta$](*v*);
    Validate[*BitwiseXorExpressionOrSuper*$^\beta$](*v*)
end proc;

Validate[*BitwiseAndExpressionOrSuper*$^\beta$]: VALIDATIONENV → ();
  Validate[*BitwiseAndExpressionOrSuper*$^\beta$ ⇒ *BitwiseAndExpression*$^\beta$] = *Validate*[*BitwiseAndExpression*$^\beta$];
  Validate[*BitwiseAndExpressionOrSuper*$^\beta$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Validate[*BitwiseXorExpressionOrSuper*^{β}]: VALIDATIONENV → ();
  Validate[*BitwiseXorExpressionOrSuper*^{β} ⇒ *BitwiseXorExpression*^{β}] = *Validate*[*BitwiseXorExpression*^{β}];
  Validate[*BitwiseXorExpressionOrSuper*^{β} ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Validate[*BitwiseOrExpressionOrSuper*^{β}]: VALIDATIONENV → ();
  Validate[*BitwiseOrExpressionOrSuper*^{β} ⇒ *BitwiseOrExpression*^{β}] = *Validate*[*BitwiseOrExpression*^{β}];
  Validate[*BitwiseOrExpressionOrSuper*^{β} ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*BitwiseAndExpression*^{β}] (*e*: DYNAMICENV): OBJORREF
  [*BitwiseAndExpression*^{β} ⇒ *EqualityExpression*^{β}] **do**
    return *Eval*[*EqualityExpression*^{β}](*e*);
  [*BitwiseAndExpression*^{β} ⇒ *BitwiseAndExpressionOrSuper*^{β} **&** *EqualityExpressionOrSuper*^{β}] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*BitwiseAndExpressionOrSuper*^{β}](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*^{β}](*e*));
    return *binaryDispatch*(*bitwiseAndTable*, *a*, *b*)
end proc;

proc *Eval*[*BitwiseXorExpression*^{β}] (*e*: DYNAMICENV): OBJORREF
  [*BitwiseXorExpression*^{β} ⇒ *BitwiseAndExpression*^{β}] **do**
    return *Eval*[*BitwiseAndExpression*^{β}](*e*);
  [*BitwiseXorExpression*^{β} ⇒ *BitwiseXorExpressionOrSuper*^{β} **^** *BitwiseAndExpressionOrSuper*^{β}] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*BitwiseXorExpressionOrSuper*^{β}](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*BitwiseAndExpressionOrSuper*^{β}](*e*));
    return *binaryDispatch*(*bitwiseXorTable*, *a*, *b*)
end proc;

proc *Eval*[*BitwiseOrExpression*^{β}] (*e*: DYNAMICENV): OBJORREF
  [*BitwiseOrExpression*^{β} ⇒ *BitwiseXorExpression*^{β}] **do**
    return *Eval*[*BitwiseXorExpression*^{β}](*e*);
  [*BitwiseOrExpression*^{β} ⇒ *BitwiseOrExpressionOrSuper*^{β} **|** *BitwiseXorExpressionOrSuper*^{β}] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*BitwiseOrExpressionOrSuper*^{β}](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*BitwiseXorExpressionOrSuper*^{β}](*e*));
    return *binaryDispatch*(*bitwiseOrTable*, *a*, *b*)
end proc;

Eval[*BitwiseAndExpressionOrSuper*^{β}]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*BitwiseAndExpressionOrSuper*^{β} ⇒ *BitwiseAndExpression*^{β}] = *Eval*[*BitwiseAndExpression*^{β}];
  Eval[*BitwiseAndExpressionOrSuper*^{β} ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Eval[*BitwiseXorExpressionOrSuper*^{β}]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*BitwiseXorExpressionOrSuper*^{β} ⇒ *BitwiseXorExpression*^{β}] = *Eval*[*BitwiseXorExpression*^{β}];
  Eval[*BitwiseXorExpressionOrSuper*^{β} ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Eval[*BitwiseOrExpressionOrSuper*^{β}]: DYNAMICENV → OBJORREFOPTIONALLIMIT;
  Eval[*BitwiseOrExpressionOrSuper*^{β} ⇒ *BitwiseOrExpression*^{β}] = *Eval*[*BitwiseOrExpression*^{β}];
  Eval[*BitwiseOrExpressionOrSuper*^{β} ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

12.18 Binary Logical Operators

Syntax

LogicalAndExpression^β ⇒
    BitwiseOrExpression^β
  | *LogicalAndExpression*^{β} **&&** *BitwiseOrExpression*^{β}

LogicalXorExpression$^\beta$ ⇒
  LogicalAndExpression$^\beta$
| *LogicalXorExpression*$^\beta$ **^^** *LogicalAndExpression*$^\beta$

LogicalOrExpression$^\beta$ ⇒
  LogicalXorExpression$^\beta$
| *LogicalOrExpression*$^\beta$ **||** *LogicalXorExpression*$^\beta$

Validation

proc *Validate*[*LogicalAndExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*LogicalAndExpression*$^\beta$ ⇒ *BitwiseOrExpression*$^\beta$] **do** *Validate*[*BitwiseOrExpression*$^\beta$](*v*);
  [*LogicalAndExpression*$^\beta{}_0$ ⇒ *LogicalAndExpression*$^\beta{}_1$ **&&** *BitwiseOrExpression*$^\beta$] **do**
    Validate[*LogicalAndExpression*$^\beta{}_1$](*v*);
    Validate[*BitwiseOrExpression*$^\beta$](*v*)
end proc;

proc *Validate*[*LogicalXorExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*LogicalXorExpression*$^\beta$ ⇒ *LogicalAndExpression*$^\beta$] **do**
    Validate[*LogicalAndExpression*$^\beta$](*v*);
  [*LogicalXorExpression*$^\beta{}_0$ ⇒ *LogicalXorExpression*$^\beta{}_1$ **^^** *LogicalAndExpression*$^\beta$] **do**
    Validate[*LogicalXorExpression*$^\beta{}_1$](*v*);
    Validate[*LogicalAndExpression*$^\beta$](*v*)
end proc;

proc *Validate*[*LogicalOrExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*LogicalOrExpression*$^\beta$ ⇒ *LogicalXorExpression*$^\beta$] **do**
    Validate[*LogicalXorExpression*$^\beta$](*v*);
  [*LogicalOrExpression*$^\beta{}_0$ ⇒ *LogicalOrExpression*$^\beta{}_1$ **||** *LogicalXorExpression*$^\beta$] **do**
    Validate[*LogicalOrExpression*$^\beta{}_1$](*v*);
    Validate[*LogicalXorExpression*$^\beta$](*v*)
end proc;

Evaluation

proc *Eval*[*LogicalAndExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*LogicalAndExpression*$^\beta$ ⇒ *BitwiseOrExpression*$^\beta$] **do**
    return *Eval*[*BitwiseOrExpression*$^\beta$](*e*);
  [*LogicalAndExpression*$^\beta{}_0$ ⇒ *LogicalAndExpression*$^\beta{}_1$ **&&** *BitwiseOrExpression*$^\beta$] **do**
    a: OBJECT ← *readReference*(*Eval*[*LogicalAndExpression*$^\beta{}_1$](*e*));
    if *toBoolean*(*a*) **then return** *readReference*(*Eval*[*BitwiseOrExpression*$^\beta$](*e*))
    else return *a*
    end if
end proc;

proc *Eval*[*LogicalXorExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*LogicalXorExpression*$^\beta$ ⇒ *LogicalAndExpression*$^\beta$] **do**
    return *Eval*[*LogicalAndExpression*$^\beta$](*e*);
  [*LogicalXorExpression*$^\beta{}_0$ ⇒ *LogicalXorExpression*$^\beta{}_1$ **^^** *LogicalAndExpression*$^\beta$] **do**
    a: OBJECT ← *readReference*(*Eval*[*LogicalXorExpression*$^\beta{}_1$](*e*));
    b: OBJECT ← *readReference*(*Eval*[*LogicalAndExpression*$^\beta$](*e*));
    ab: BOOLEAN ← *toBoolean*(*a*);
    bb: BOOLEAN ← *toBoolean*(*b*);
    return *ab* **xor** *bb*
end proc;

proc *Eval*[*LogicalOrExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*LogicalOrExpression*$^\beta$ ⇒ *LogicalXorExpression*$^\beta$] **do**
    return *Eval*[*LogicalXorExpression*$^\beta$](*e*);
  [*LogicalOrExpression*$^\beta{}_0$ ⇒ *LogicalOrExpression*$^\beta{}_1$ **||** *LogicalXorExpression*$^\beta$] **do**
    a: OBJECT ← *readReference*(*Eval*[*LogicalOrExpression*$^\beta{}_1$](*e*));
    if *toBoolean*(*a*) **then return** *a*
    else return *readReference*(*Eval*[*LogicalXorExpression*$^\beta$](*e*))
    end if
end proc;

12.19 Conditional Operator

Syntax

ConditionalExpression$^\beta$ ⇒
  LogicalOrExpression$^\beta$
| *LogicalOrExpression*$^\beta$ **?** *AssignmentExpression*$^\beta$ **:** *AssignmentExpression*$^\beta$

NonAssignmentExpression$^\beta$ ⇒
  LogicalOrExpression$^\beta$
| *LogicalOrExpression*$^\beta$ **?** *NonAssignmentExpression*$^\beta$ **:** *NonAssignmentExpression*$^\beta$

Validation

proc *Validate*[*ConditionalExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*ConditionalExpression*$^\beta$ ⇒ *LogicalOrExpression*$^\beta$] **do**
    Validate[*LogicalOrExpression*$^\beta$](*v*);
  [*ConditionalExpression*$^\beta$ ⇒ *LogicalOrExpression*$^\beta$ **?** *AssignmentExpression*$^\beta{}_1$ **:** *AssignmentExpression*$^\beta{}_2$] **do**
    Validate[*LogicalOrExpression*$^\beta$](*v*);
    Validate[*AssignmentExpression*$^\beta{}_1$](*v*);
    Validate[*AssignmentExpression*$^\beta{}_2$](*v*)
end proc;

Evaluation

proc *Eval*[*ConditionalExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*ConditionalExpression*$^\beta$ ⇒ *LogicalOrExpression*$^\beta$] **do**
    return *Eval*[*LogicalOrExpression*$^\beta$](*e*);
  [*ConditionalExpression*$^\beta$ ⇒ *LogicalOrExpression*$^\beta$ **?** *AssignmentExpression*$^\beta{}_1$ **:** *AssignmentExpression*$^\beta{}_2$] **do**
    if *toBoolean*(*readReference*(*Eval*[*LogicalOrExpression*$^\beta$](*e*))) **then**
      return *Eval*[*AssignmentExpression*$^\beta{}_1$](*e*)
    else return *Eval*[*AssignmentExpression*$^\beta{}_2$](*e*)
    end if
end proc;

12.20 Assignment Operators

Syntax

AssignmentExpression$^\beta$ ⇒
  ConditionalExpression$^\beta$
| *PostfixExpression* **=** *AssignmentExpression*$^\beta$
| *PostfixExpressionOrSuper CompoundAssignment AssignmentExpression*$^\beta$
| *PostfixExpressionOrSuper CompoundAssignment SuperExpression*
| *PostfixExpression LogicalAssignment AssignmentExpression*$^\beta$

CompoundAssignment ⇒
  `*=`
| **`/=`**
| **`%=`**
| **`+=`**
| **`-=`**
| **`<<=`**
| **`>>=`**
| **`>>>=`**
| **`&=`**
| **`^=`**
| **`|=`**

LogicalAssignment ⇒
  `&&=`
| **`^^=`**
| **`||=`**

Validation

proc *Validate*[*AssignmentExpression*$^\beta$] (*v*: VALIDATIONENV)
  [*AssignmentExpression*$^\beta$ ⇒ *ConditionalExpression*$^\beta$] **do**
    Validate[*ConditionalExpression*$^\beta$](*v*);
  [*AssignmentExpression*$^\beta{}_0$ ⇒ *PostfixExpression* **=** *AssignmentExpression*$^\beta{}_1$] **do**
    Validate[*PostfixExpression*](*v*);
    Validate[*AssignmentExpression*$^\beta{}_1$](*v*);
  [*AssignmentExpression*$^\beta{}_0$ ⇒ *PostfixExpressionOrSuper CompoundAssignment AssignmentExpression*$^\beta{}_1$] **do**
    Validate[*PostfixExpressionOrSuper*](*v*);
    Validate[*AssignmentExpression*$^\beta{}_1$](*v*);
  [*AssignmentExpression*$^\beta$ ⇒ *PostfixExpressionOrSuper CompoundAssignment SuperExpression*] **do**
    Validate[*PostfixExpressionOrSuper*](*v*);
    Validate[*SuperExpression*](*v*);
  [*AssignmentExpression*$^\beta{}_0$ ⇒ *PostfixExpression LogicalAssignment AssignmentExpression*$^\beta{}_1$] **do**
    Validate[*PostfixExpression*](*v*);
    Validate[*AssignmentExpression*$^\beta{}_1$](*v*)
end proc;

Evaluation

proc *Eval*[*AssignmentExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
  [*AssignmentExpression*$^\beta$ ⇒ *ConditionalExpression*$^\beta$] **do**
    return *Eval*[*ConditionalExpression*$^\beta$](*e*);
  [*AssignmentExpression*$^\beta{}_0$ ⇒ *PostfixExpression* **=** *AssignmentExpression*$^\beta{}_1$] **do**
    r: OBJORREF ← *Eval*[*PostfixExpression*](*e*);
    a: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^\beta{}_1$](*e*));
    writeReference(*r*, *a*);
    return *a*;
  [*AssignmentExpression*$^\beta{}_0$ ⇒ *PostfixExpressionOrSuper CompoundAssignment AssignmentExpression*$^\beta{}_1$] **do**
    return *evalAssignmentOp*(*Table*[*CompoundAssignment*], *Eval*[*PostfixExpressionOrSuper*],
        Eval[*AssignmentExpression*$^\beta{}_1$], *e*);
  [*AssignmentExpression*$^\beta$ ⇒ *PostfixExpressionOrSuper CompoundAssignment SuperExpression*] **do**
    return *evalAssignmentOp*(*Table*[*CompoundAssignment*], *Eval*[*PostfixExpressionOrSuper*], *Eval*[*SuperExpression*],
        e);

[*AssignmentExpression*$^\beta$ ⇒ *PostfixExpression LogicalAssignment AssignmentExpression*$^\beta$] **do**
????
end proc;

Table[*CompoundAssignment*]: BINARYMETHOD{};
Table[*CompoundAssignment* ⇒ ***=**] = *multiplyTable*;
Table[*CompoundAssignment* ⇒ **/=**] = *divideTable*;
Table[*CompoundAssignment* ⇒ **%=**] = *remainderTable*;
Table[*CompoundAssignment* ⇒ **+=**] = *addTable*;
Table[*CompoundAssignment* ⇒ **-=**] = *subtractTable*;
Table[*CompoundAssignment* ⇒ **<<=**] = *shiftLeftTable*;
Table[*CompoundAssignment* ⇒ **>>=**] = *shiftRightTable*;
Table[*CompoundAssignment* ⇒ **>>>=**] = *shiftRightUnsignedTable*;
Table[*CompoundAssignment* ⇒ **&=**] = *bitwiseAndTable*;
Table[*CompoundAssignment* ⇒ **^=**] = *bitwiseXorTable*;
Table[*CompoundAssignment* ⇒ **|=**] = *bitwiseOrTable*;

proc *evalAssignmentOp*(*table*: BINARYMETHOD{}, *leftEval*: DYNAMICENV → OBJORREFOPTIONALLIMIT,
rightEval: DYNAMICENV → OBJORREFOPTIONALLIMIT, *e*: DYNAMICENV): OBJORREF
rLeft: OBJORREFOPTIONALLIMIT ← *leftEval*(*e*);
oLeft: OBJOPTIONALLIMIT ← *readRefWithLimit*(*rLeft*);
oRight: OBJOPTIONALLIMIT ← *readRefWithLimit*(*rightEval*(*e*));
result: OBJECT ← *binaryDispatch*(*table*, *oLeft*, *oRight*);
writeReference(*rLeft*, *result*);
return *result*
end proc;

12.21 Comma Expressions

Syntax

ListExpression$^\beta$ ⇒
AssignmentExpression$^\beta$
| *ListExpression*$^\beta$ **,** *AssignmentExpression*$^\beta$

OptionalExpression ⇒
ListExpression$^{\text{allowIn}}$
| «empty»

Validation

proc *Validate*[*ListExpression*$^\beta$] (*v*: VALIDATIONENV)
[*ListExpression*$^\beta$ ⇒ *AssignmentExpression*$^\beta$] **do** *Validate*[*AssignmentExpression*$^\beta$](*v*);
[*ListExpression*$^\beta{}_0$ ⇒ *ListExpression*$^\beta{}_1$ **,** *AssignmentExpression*$^\beta$] **do**
Validate[*ListExpression*$^\beta{}_1$](*v*);
Validate[*AssignmentExpression*$^\beta$](*v*)
end proc;

Evaluation

proc *Eval*[*ListExpression*$^\beta$] (*e*: DYNAMICENV): OBJORREF
[*ListExpression*$^\beta$ ⇒ *AssignmentExpression*$^\beta$] **do return** *Eval*[*AssignmentExpression*$^\beta$](*e*);
[*ListExpression*$^\beta{}_0$ ⇒ *ListExpression*$^\beta{}_1$ **,** *AssignmentExpression*$^\beta$] **do**
readReference(*Eval*[*ListExpression*$^\beta{}_1$](*e*));
return *readReference*(*Eval*[*AssignmentExpression*$^\beta$](*e*))
~~**return** *Eval*[*AssignmentExpression*$^\beta$](*e*)~~
end proc;

proc *EvalAsList*[*ListExpression*$^\beta$] (*e*: DYNAMICENV): OBJECT[]
  [*ListExpression*$^\beta$ ⇒ *AssignmentExpression*$^\beta$] **do**
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^\beta$](*e*));
    return [*elt***]**;
  [*ListExpression*$^\beta_0$ ⇒ *ListExpression*$^\beta_1$ **,** *AssignmentExpression*$^\beta$] **do**
    elts: OBJECT[] ← *EvalAsList*[*ListExpression*$^\beta_1$](*e*);
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^\beta$](*e*));
    return *elts* ⊕ **[***elt***]**
end proc;

12.22 Type Expressions

Syntax

TypeExpression$^\beta$ ⇒ *NonAssignmentExpression*$^\beta$

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

19.1 Unary Operators

proc *plusObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT
  return *toNumber*(*a*)
end proc;

proc *minusObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT
  return *float64Negate*(*toNumber*(*a*))
end proc;

proc *bitwiseNotObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT
  i: INTEGER ← *toInt32*(*toNumber*(*a*));
  return *realToFloat64*(*bitwiseXor*(*i*, –1))
end proc;

proc *incrementObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT
  x: OBJECT ← *unaryPlus*(*a*);
  return *binaryDispatch*(*addTable*, *x*, 1.0)
end proc;

```
proc decrementObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   x: OBJECT ← unaryPlus(a);
   return binaryDispatch(subtractTable, x, 1.0)
end proc;

proc callObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   case a of
      UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE ∪ PROTOTYPE do
         throw typeError;
      CLASS ∪ INSTANCE do return a.call(this, args);
      METHODCLOSURE do return callObject(a.this, a.method.f, args)
   end case
end proc;

proc constructObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   case a of
      UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE ∪ METHODCLOSURE ∪
            PROTOTYPE do
         throw typeError;
      CLASS ∪ INSTANCE do return a.construct(this, args)
   end case
end proc;

proc bracketReadObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
   name: STRING ← toString(args.positional[0]);
   return readQualifiedProperty(a, QUALIFIEDNAME⟨publicNamespace, name⟩, true)
end proc;

proc bracketWriteObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   if |args.positional| ≠ 2 or args.named ≠ {} then throw argumentMismatchError end if;
   newValue: OBJECT ← args.positional[0];
   name: STRING ← toString(args.positional[1]);
   writeQualifiedProperty(a, QUALIFIEDNAME⟨publicNamespace, name⟩, true, newValue);
   return undefined
end proc;

proc bracketDeleteObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
   if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
   name: STRING ← toString(args.positional[0]);
   return deleteQualifiedProperty(a, name, publicNamespace, true)
end proc;
```

~~proc plusObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT~~
~~return toNumber(a)~~
~~end proc;~~

~~proc minusObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT~~
~~return float64Negate(toNumber(a))~~
~~end proc;~~

~~proc bitwiseNotObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT~~
~~i: INTEGER ← toInt32(toNumber(a));~~
~~return realToFloat64(bitwiseXor(i, –1))~~
~~end proc;~~

~~proc incrementObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT~~
~~x: OBJECT ← unaryPlus(a);~~
~~return binaryDispatch(addTable, null, null, x, 1.0)~~
~~end proc;~~

~~**proc** *decrementObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~*x*: OBJECT ← *unaryPlus*(*a*);~~
~~**return** *binaryDispatch*(*subtractTable*, **null**, **null**, *x*, 1.0)~~
~~**end proc**;~~

~~**proc** *callObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~**case** *a* **of**~~
~~UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE **do**~~
~~**throw typeError**;~~
~~CLASS ∪ INSTANCE **do return** *a*.call(*this*, *args*);~~
~~METHODCLOSURE **do return** *callObject*(*a*.this, *a*.method.f, *args*)~~
~~**end case**~~
~~**end proc**;~~

~~**proc** *constructObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~**case** *a* **of**~~
~~UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE ∪ METHODCLOSURE **do**~~
~~**throw typeError**;~~
~~CLASS ∪ INSTANCE **do return** *a*.construct(*this*, *args*)~~
~~**end case**~~
~~**end proc**;~~

~~**proc** *bracketReadObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~**if** |*args*.positional| ≠ 1 **or** *args*.named ≠ {} **then throw argumentMismatchError end if**;~~
~~*name*: STRING ← *toString*(*args*.positional[0]);~~
~~**return** *readQualifiedProperty*(*a*, *name*, *publicNamespace*, **true**)~~
~~**end proc**;~~

~~**proc** *bracketWriteObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~**if** |*args*.positional| ≠ 2 **or** *args*.named ≠ {} **then throw argumentMismatchError end if**;~~
~~*newValue*: OBJECT ← *args*.positional[0];~~
~~*name*: STRING ← *toString*(*args*.positional[1]);~~
~~*writeQualifiedProperty*(*a*, *name*, *publicNamespace*, **true**, *newValue*);~~
~~**return undefined**~~
~~**end proc**;~~

~~**proc** *bracketDeleteObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST): OBJECT~~
~~**if** |*args*.positional| ≠ 1 **or** *args*.named ≠ {} **then throw argumentMismatchError end if**;~~
~~*name*: STRING ← *toString*(*args*.positional[0]);~~
~~**return** *deleteQualifiedProperty*(*a*, *name*, *publicNamespace*, **true**)~~
~~**end proc**;~~

plusTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *plusObject*⟩};

minusTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *minusObject*⟩};

bitwiseNotTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *bitwiseNotObject*⟩};

incrementTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *incrementObject*⟩};

decrementTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *decrementObject*⟩};

callTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *callObject*⟩};

constructTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *constructObject*⟩};

bracketReadTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *bracketReadObject*⟩};

bracketWriteTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *bracketWriteObject*⟩};

bracketDeleteTable: UNARYMETHOD{} ← {UNARYMETHOD⟨*objectClass*, *bracketDeleteObject*⟩};

~~*plusTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *plusObject*⟩}⟩⟩;~~

~~*minusTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *minusObject*⟩}⟩⟩;~~

~~*bitwiseNotTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *bitwiseNotObject*⟩}⟩⟩;~~

~~*incrementTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *incrementObject*⟩}⟩⟩;~~

~~*decrementTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *decrementObject*⟩}⟩⟩;~~

~~*callTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *callObject*⟩}⟩⟩;~~

~~*constructTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *constructObject*⟩}⟩⟩;~~

~~*bracketReadTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *bracketReadObject*⟩}⟩⟩;~~

~~*bracketWriteTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *bracketWriteObject*⟩}⟩⟩;~~

~~*bracketDeleteTable*: UNARYTABLE = **new** UNARYTABLE⟨⟨{UNARYMETHOD⟨*objectClass*, *bracketDeleteObject*⟩}⟩⟩;~~

19.2 Binary Operators

proc *addObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    ap: OBJECT ← *toPrimitive*(*a*, **null**);
    bp: OBJECT ← *toPrimitive*(*b*, **null**);
    if *ap* ∈ STRING **or** *bp* ∈ STRING **then return** *toString*(*ap*) ⊕ *toString*(*bp*)
    else return *float64Add*(*toNumber*(*ap*), *toNumber*(*bp*))
    end if
end proc;

proc *subtractObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    return *float64Subtract*(*toNumber*(*a*), *toNumber*(*b*))
end proc;

proc *multiplyObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    return *float64Multiply*(*toNumber*(*a*), *toNumber*(*b*))
end proc;

proc *divideObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    return *float64Divide*(*toNumber*(*a*), *toNumber*(*b*))
end proc;

proc *remainderObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    return *float64Remainder*(*toNumber*(*a*), *toNumber*(*b*))
end proc;

proc *lessObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
    ap: OBJECT ← *toPrimitive*(*a*, **null**);
    bp: OBJECT ← *toPrimitive*(*b*, **null**);
    if *ap* ∈ STRING **and** *bp* ∈ STRING **then return** *ap* < *bp*
    else return *float64Compare*(*toNumber*(*ap*), *toNumber*(*bp*)) = **less**
    end if
end proc;

```
proc lessOrEqualObjects(a: OBJECT, b: OBJECT): OBJECT
   ap: OBJECT ← toPrimitive(a, null);
   bp: OBJECT ← toPrimitive(b, null);
   if ap ∈ STRING and bp ∈ STRING then return ap ≤ bp
   else return float64Compare(toNumber(ap), toNumber(bp)) ∈ {less, equal}
   end if
end proc;

proc equalObjects(a: OBJECT, b: OBJECT): OBJECT
   case a of
      UNDEFINED ∪ NULL do return b ∈ UNDEFINED ∪ NULL;
      BOOLEAN do
         if b ∈ BOOLEAN then return a = b
         else return equalObjects(toNumber(a), b)
         end if;
      FLOAT64 do
         bp: OBJECT ← toPrimitive(b, null);
         case bp of
            UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪
                  INSTANCE do
               return false;
            BOOLEAN ∪ STRING ∪ FLOAT64 do
               return float64Compare(a, toNumber(bp)) = equal
         end case;
      STRING do
         bp: OBJECT ← toPrimitive(b, null);
         case bp of
            UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪
                  INSTANCE do
               return false;
            BOOLEAN ∪ FLOAT64 do
               return float64Compare(toNumber(a), toNumber(bp)) = equal;
            STRING do return a = bp
         end case;
      NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do
         case b of
            UNDEFINED ∪ NULL do return false;
            NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do
               return strictEqualObjects(a, b);
            BOOLEAN ∪ FLOAT64 ∪ STRING do
               ap: OBJECT ← toPrimitive(a, null);
               case ap of
                  UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪
                        INSTANCE do
                     return false;
                  BOOLEAN ∪ FLOAT64 ∪ STRING do return equalObjects(ap, b)
               end case
         end case
   end case
end proc;

proc strictEqualObjects(a: OBJECT, b: OBJECT): OBJECT
   if a ∈ FLOAT64 and b ∈ FLOAT64 then return float64Compare(a, b) = equal
   else return a = b
   end if
end proc;
```

proc *shiftLeftObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toUInt32*(*toNumber*(*a*));
  count: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);
  return *realToFloat64*(*uInt32ToInt32*(*bitwiseAnd*(*bitwiseShift*(*i*, *count*), 0xFFFFFFFF)))
end proc;

proc *shiftRightObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toInt32*(*toNumber*(*a*));
  count: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);
  return *realToFloat64*(*bitwiseShift*(*i*, –*count*))
end proc;

proc *shiftRightUnsignedObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toUInt32*(*toNumber*(*a*));
  count: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);
  return *realToFloat64*(*bitwiseShift*(*i*, –*count*))
end proc;

proc *bitwiseAndObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toInt32*(*toNumber*(*a*));
  j: INTEGER ← *toInt32*(*toNumber*(*b*));
  return *realToFloat64*(*bitwiseAnd*(*i*, *j*))
end proc;

proc *bitwiseXorObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toInt32*(*toNumber*(*a*));
  j: INTEGER ← *toInt32*(*toNumber*(*b*));
  return *realToFloat64*(*bitwiseXor*(*i*, *j*))
end proc;

proc *bitwiseOrObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT
  i: INTEGER ← *toInt32*(*toNumber*(*a*));
  j: INTEGER ← *toInt32*(*toNumber*(*b*));
  return *realToFloat64*(*bitwiseOr*(*i*, *j*))
end proc;

~~**proc** *addObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
  ~~*ap*: OBJECT ← *toPrimitive*(*a*, **null**);~~
  ~~*bp*: OBJECT ← *toPrimitive*(*b*, **null**);~~
  ~~**if** *ap* ∈ STRING **or** *bp* ∈ STRING **then return** *toString*(*ap*) ⊕ *toString*(*bp*)~~
  ~~**else return** *float64Add*(*toNumber*(*ap*), *toNumber*(*bp*))~~
  ~~**end if**~~
~~**end proc**;~~

~~**proc** *subtractObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
  ~~**return** *float64Subtract*(*toNumber*(*a*), *toNumber*(*b*))~~
~~**end proc**;~~

~~**proc** *multiplyObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
  ~~**return** *float64Multiply*(*toNumber*(*a*), *toNumber*(*b*))~~
~~**end proc**;~~

~~**proc** *divideObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
  ~~**return** *float64Divide*(*toNumber*(*a*), *toNumber*(*b*))~~
~~**end proc**;~~

~~**proc** *remainderObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
  ~~**return** *float64Remainder*(*toNumber*(*a*), *toNumber*(*b*))~~
~~**end proc**;~~

```
proc lessObjects(a: OBJECT, b: OBJECT): OBJECT
   ap: OBJECT ← toPrimitive(a, null);
   bp: OBJECT ← toPrimitive(b, null);
   if ap ∈ STRING and bp ∈ STRING then return ap < bp
   else return float64Compare(toNumber(ap), toNumber(bp)) = less
   end if
end proc;

proc lessOrEqualObjects(a: OBJECT, b: OBJECT): OBJECT
   ap: OBJECT ← toPrimitive(a, null);
   bp: OBJECT ← toPrimitive(b, null);
   if ap ∈ STRING and bp ∈ STRING then return ap ≤ bp
   else return float64Compare(toNumber(ap), toNumber(bp)) ∈ {less, equal}
   end if
end proc;

proc equalObjects(a: OBJECT, b: OBJECT): OBJECT
   case a of
      UNDEFINED ∪ NULL do return b ∈ UNDEFINED ∪ NULL;
      BOOLEAN do
         if b ∈ BOOLEAN then return a = b
         else return equalObjects(toNumber(a), b)
         end if;
      FLOAT64 do
         bp: OBJECT ← toPrimitive(b, null);
         case bp of
            UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE do
               return false;
            BOOLEAN ∪ STRING ∪ FLOAT64 do
               return float64Compare(a, toNumber(bp)) = equal
         end case;
      STRING do
         bp: OBJECT ← toPrimitive(b, null);
         case bp of
            UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE do
               return false;
            BOOLEAN ∪ FLOAT64 do
               return float64Compare(toNumber(a), toNumber(bp)) = equal;
            STRING do return a = bp
         end case;
      NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE do
         case b of
            UNDEFINED ∪ NULL do return false;
            NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE do
               return strictEqualObjects(a, b);
            BOOLEAN ∪ FLOAT64 ∪ STRING do
               ap: OBJECT ← toPrimitive(a, null);
               case ap of
                  UNDEFINED ∪ NULL ∪ NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ INSTANCE do
                     return false;
                  BOOLEAN ∪ FLOAT64 ∪ STRING do return equalObjects(ap, b)
               end case
         end case
   end case
end proc;
```

~~**proc** *strictEqualObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~**if** *a* ∈ FLOAT64 **and** *b* ∈ FLOAT64 **then return** *float64Compare*(*a*, *b*) = **equal**~~
~~**else return** *a* = *b*~~
~~**end if**~~
~~**end proc**;~~

~~**proc** *shiftLeftObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toUInt32*(*toNumber*(*a*));~~
~~*count*: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);~~
~~**return** *realToFloat64*(*uInt32ToInt32*(*bitwiseAnd*(*bitwiseShift*(*i*, *count*), 0xFFFFFFFF)))~~
~~**end proc**;~~

~~**proc** *shiftRightObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toInt32*(*toNumber*(*a*));~~
~~*count*: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);~~
~~**return** *realToFloat64*(*bitwiseShift*(*i*, −*count*))~~
~~**end proc**;~~

~~**proc** *shiftRightUnsignedObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toUInt32*(*toNumber*(*a*));~~
~~*count*: INTEGER ← *bitwiseAnd*(*toUInt32*(*toNumber*(*b*)), 0x1F);~~
~~**return** *realToFloat64*(*bitwiseShift*(*i*, −*count*))~~
~~**end proc**;~~

~~**proc** *bitwiseAndObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toInt32*(*toNumber*(*a*));~~
~~*j*: INTEGER ← *toInt32*(*toNumber*(*b*));~~
~~**return** *realToFloat64*(*bitwiseAnd*(*i*, *j*))~~
~~**end proc**;~~

~~**proc** *bitwiseXorObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toInt32*(*toNumber*(*a*));~~
~~*j*: INTEGER ← *toInt32*(*toNumber*(*b*));~~
~~**return** *realToFloat64*(*bitwiseXor*(*i*, *j*))~~
~~**end proc**;~~

~~**proc** *bitwiseOrObjects*(*a*: OBJECT, *b*: OBJECT): OBJECT~~
~~*i*: INTEGER ← *toInt32*(*toNumber*(*a*));~~
~~*j*: INTEGER ← *toInt32*(*toNumber*(*b*));~~
~~**return** *realToFloat64*(*bitwiseOr*(*i*, *j*))~~
~~**end proc**;~~

addTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *addObjects*⟩};

subtractTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *subtractObjects*⟩};

multiplyTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *multiplyObjects*⟩};

divideTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *divideObjects*⟩};

remainderTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *remainderObjects*⟩};

lessTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *lessObjects*⟩};

lessOrEqualTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *lessOrEqualObjects*⟩};

equalTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *equalObjects*⟩};

strictEqualTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *strictEqualObjects*⟩};

shiftLeftTable: BINARYMETHOD{} ← {BINARYMETHOD⟨*objectClass*, *objectClass*, *shiftLeftObjects*⟩};

shiftRightTable: BinaryMethod{} ← {BinaryMethod⟨*objectClass*, *objectClass*, *shiftRightObjects*⟩};

shiftRightUnsignedTable: BinaryMethod{}
    ← {BinaryMethod⟨*objectClass*, *objectClass*, *shiftRightUnsignedObjects*⟩};

bitwiseAndTable: BinaryMethod{} ← {BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseAndObjects*⟩};

bitwiseXorTable: BinaryMethod{} ← {BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseXorObjects*⟩};

bitwiseOrTable: BinaryMethod{} ← {BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseOrObjects*⟩};

~~*addTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *addObjects*⟩}⟫;~~

~~*subtractTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *subtractObjects*⟩}⟫;~~

~~*multiplyTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *multiplyObjects*⟩}⟫;~~

~~*divideTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *divideObjects*⟩}⟫;~~

~~*remainderTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *remainderObjects*⟩}⟫;~~

~~*lessTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *lessObjects*⟩}⟫;~~

~~*lessOrEqualTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *lessOrEqualObjects*⟩}⟫;~~

~~*equalTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *equalObjects*⟩}⟫;~~

~~*strictEqualTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *strictEqualObjects*⟩}⟫;~~

~~*shiftLeftTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *shiftLeftObjects*⟩}⟫;~~

~~*shiftRightTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *shiftRightObjects*⟩}⟫;~~

~~*shiftRightUnsignedTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *shiftRightUnsignedObjects*⟩}⟫;~~

~~*bitwiseAndTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseAndObjects*⟩}⟫;~~

~~*bitwiseXorTable*: BinaryTable~~
    ~~= **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseXorObjects*⟩}⟫;~~

~~*bitwiseOrTable*: BinaryTable = **new** BinaryTable⟪{BinaryMethod⟨*objectClass*, *objectClass*, *bitwiseOrObjects*⟩}⟫;~~

20 Built-in Namespaces

21 Built-in Units

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

23.3 Units

A Index

A.1 Nonterminals

A.2 Tags

A.3 ~~Types~~Semantic Domains

A.4 Globals