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.

Chapters 10 and 12–16 have been revised extensively to add the character and long integer classes, remove units and operator overriding, and implement more of the semantics. These chapters do not have change bars.

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*. BOOLEAN is the two-element semantic domain {**true**, **false**}.

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

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

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

5.5 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:

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

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

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

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

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

A set can also be written using the set comprehension notation

$\{f(x) \mid \square x \square A\}$

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

$\{f(x) \mid \square x \square A$ **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,

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

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:

$x \square A$	**true** if x is an element of A and **false** if not
$x \square A$	**false** if x is an element of A and **true** if not
$\lvert A \rvert$	The number of elements in A (only used on finite sets)
min A	The value m that satisfies both $m \square A$ and for all elements $x \square A$, $x \geq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
max A	The value m that satisfies both $m \square A$ and for all elements $x \square A$, $x \leq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
$A \square B$	The intersection of A and B (the set or semantic domain of all values that are present both in A and in B)
$A \square B$	The union of A and B (the set or semantic domain of all values that are present in at least one of A or B)
$A - B$	The difference of A and B (the set or semantic domain of all values that are present in A but not B)
$A = B$	**true** if A and B are equal and **false** otherwise. 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 A and B are equal and **true** otherwise
$A \square B$	**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.
$A \square B$	**true** if A is a proper subset of B and **false** otherwise. $A \square B$ is equivalent to $A \square B$ **and** $A \neq B$.

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

T = {1,2,3}

then:

T{} = {{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}

The empty set {} is a member of T{} for any semantic domain T.

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

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

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

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

The quantifier

every x ∈ 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 ∈ {3, 16, 19, 26} **satisfies** x **mod** 10 = 6) = **false**;
(**every** x ∈ {6, 26, 96, 106} **satisfies** x **mod** 10 = 6) = **true**;
(**every** x ∈ {} **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 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 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 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 yth power (used only when either $x \neq 0$ and y is an integer or x is any number and $y > 0$)
$\lvert x \rvert$	The absolute value of x, which is x if $x \geq 0$ and $-x$ otherwise
$\lfloor x \rfloor$	*Floor* of x, which is the unique integer i such that $i \leq x < i+1$. $\lfloor 3 \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, and $\lfloor 7 \rfloor = 7$.
$\lceil x \rceil$	*Ceiling* of x, which is the unique integer i such that $i-1 < x \leq i$. $\lceil 3 \rceil = 4$, $\lceil -3.5 \rceil = -3$, 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 ∈ {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

$a_i = \lfloor x / 2^i \rfloor$ **mod** 2

If x is zero or positive, then its sequence will have infinitely many consecutive leading 0's, while a negative integer x will generate a sequence with infinitely many consecutive leading 1's. For example, 6 generates the sequence ...0...0000110, while –6 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 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 $\lfloor x \cdot 2^{count} \rfloor$

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 ∪ DENORMALISED~~NONZEROFINITEFLOAT64 ∪ {**+zero**, **–zero**};
NONZEROFINITEFLOAT64 = NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64;

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

NORMALISEDFLOAT64 = $\{s \cdot m \cdot 2^e \mid \lfloor s \rfloor \in \{-1, 1\}, \lfloor m \rfloor \in \{2^{52} \ldots 2^{53}-1\}, \lfloor e \rfloor \in \{-1074 \ldots 971\}\}$

m is called the *significand*.

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

DENORMALISEDFLOAT64 = $\{s \cdot m \cdot 2^{-1074} \mid \lfloor s \rfloor \in \{-1, 1\}, \lfloor m \rfloor \in \{1 \ldots 2^{52}-1\}\}$

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 NONZEROFINITEFLOAT64~~NORMALISEDFLOAT64 ∪ DENORMALISEDFLOAT64~~ that are greater than zero are called *positive finite*. The remaining members of NONZEROFINITEFLOAT64~~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{} ∪ NONZEROFINITEFLOAT64~~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 $a \neq 0$ **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** ⌊x⌋ **else return** ⌈x⌉ **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

	y						
x	**–∞**	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

	y						
x	**–∞**	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

	y						
x	**–∞**	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**

5.8 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 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**.

5.9 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

[$f(x)$ | x ∈ u**]**

which denotes the list **[**$f(u[0]), f(u[1]), \ldots, f(u[|u|-1])$**]** 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:

[$f(x)$ | x ∈ u **such that** *predicate*(x)**]**

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,

[x^2 | x ∈ **[**–1, 1, 2, 3, 4, 2, 5**]] = [**1, 1, 4, 9, 16, 4, 25**]**

[x+1 | x ∈ **[**–1, 1, 2, 3, 4, 5, 3, 10**] such that** x **mod** 2 = 1**] = [**0, 2, 4, 6, 4**]**

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

Notation	Precondition	Description
$\|u\|$		The length n of the list
$u[i]$	$0 \le i < \|u\|$	The ith 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 ith and the jth, 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 ith 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 ith 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, then T[] is the semantic domain of all lists whose elements are members of T. The empty list **[]** is a member of T[] for any semantic domain T.

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

some x ∈ u **satisfies** *predicate*(x)
every x ∈ 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 ∈ **[**3, 36, 19, 26**] satisfies** x **mod** 10 = 6

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

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 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 of all strings. STRING = CHARACTER[].

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 domains of possible values that the corresponding fields may hold.

The notation

NAME[label_1: v_1, ... , label_n: v_n]

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 T_i. When most of the fields are copied from an existing tuple a, this notation can be abbreviated as

NAME[label_{i1}: v_{i1}, ... , label_{ik}: v_{ik}, other fields from a]

which represents a tuple with name NAME and values v_{i1} through v_{ik} for fields labeled label_{i1} through label_{ik} respectively and the values of correspondingly labeled fields from a for all other fields.

If a is the tuple NAME[label_1: v_1, ... , label_n: v_n] then

a.label_i

returns the ith field’s value v_i.

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

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

5.12 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 domains of possible values that the corresponding fields may hold.

The expression

new NAME[ ][$label_1$: v_1, ... , $label_n$: v_n]

creates a record with name NAME and a new address . The fields labelled $label_1$ through $label_n$ at address are initialised with values v_1 through v_n respectively. Each value v_i is a member of the corresponding semantic domain T_i. A $label_k$: v_k pair may be omitted from a **new** expression, which indicates that the initial value of field $label_k$ does not matter because the semantics will always explicitly write a value into that field before reading it.

When most of the fields are copied from an existing record a, the **new** expression can be abbreviated as

new NAME[ ][$label_{i1}$: v_{i1}, ... , $label_{ik}$: v_{ik}, other fields from a]

which represents a record b with name NAME and a new address . The fields labeled $label_{i1}$ through $label_{ik}$ at address are initialised with values v_{i1} through v_{ik} respectively; the other fields at address are initialised with the values of correspondingly labeled fields from a's address.

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

a.$label_i$

returns the current value v of the ith field at address . That field may be set to a new value w, which must be a member of the semantic domain T_i, using the assignment

a.$label_i$ w

after which a.$label_i$ will evaluate to w. Any record with a different address is unaffected by the assignment.

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

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

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, ... , $param_n$: 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 semantic domains, T is the semantic domain 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.

5.13.1 Operations

The only operation done on a procedure f is calling it using the f(arg_1, ..., 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.

5.13.2 Semantic Domains of Procedures

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

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 temporary variable is referenced in a procedure, the variable's semantic domain T is listed; any value stored in *v* is guaranteed to be a member of the semantic domain T.

v: T

This step declares *v* to be a temporary variable with semantic domain T without assigning anything to the variable. *v* will not be read unless some other step first assigns a value to it.

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 **do** *step*; *step*; ...; *step*;
  T_2 **do** *step*; *step*; ...; *step*;
  ...;
  T_n **do** *step*; *step*; ...; *step*
  else *step*; *step*; ...; *step*
end case

A **case** step computes *expression*, which will evaluate to a value v. If $v \in T_1$, then the first list of *step*s is performed. Otherwise, if $v \in T_2$, then the second list of *step*s is performed, and so on. If v is not a member of any T_i, 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.

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).

for each $x \in$ *expression* **do**
  step;
  step;
  ...;
  step
end for each

A **for each** step computes *expression*, which will evaluate to either a set or a list A. The list of *step*s is performed repeatedly with variable x bound to each element of A. If A is a list, x is bound to each of its elements in order; if A is a set, the order in which x is bound to its elements is arbitrary. The repetition ends after x has been bound to all elements of A (or when either 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 **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, 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, 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.

5.13.4 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.

5.14 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 □ that contains a nonterminal *N*, one may replace an occurrence of *N* in □ 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.

5.14.1 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*.

5.14.2 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.

5.14.3 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*^{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*^{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.

5.14.4 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

[] ∈ {normal, initial}

□□ {allowIn, noIn}

introduce grammar arguments □ and □. 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□,□ □
  ConditionalExpression□,□
| *LeftSideExpression*□ = *AssignmentExpression*normal,□
| *LeftSideExpression*□ *CompoundAssignment AssignmentExpression*normal,□

expands into the following four rules:

*AssignmentExpression*normal,allowIn □
  *ConditionalExpression*normal,allowIn
| *LeftSideExpression*normal = *AssignmentExpression*normal,allowIn
| *LeftSideExpression*normal *CompoundAssignment AssignmentExpression*normal,allowIn

*AssignmentExpression*normal,noIn □
  *ConditionalExpression*normal,noIn
| *LeftSideExpression*normal = *AssignmentExpression*normal,noIn
| *LeftSideExpression*normal *CompoundAssignment AssignmentExpression*normal,noIn

*AssignmentExpression*initial,allowIn □
  *ConditionalExpression*initial,allowIn
| *LeftSideExpression*initial = *AssignmentExpression*normal,allowIn
| *LeftSideExpression*initial *CompoundAssignment AssignmentExpression*normal,allowIn

*AssignmentExpression*initial,noIn □
  *ConditionalExpression*initial,noIn
| *LeftSideExpression*initial = *AssignmentExpression*normal,noIn
| *LeftSideExpression*initial *CompoundAssignment AssignmentExpression*normal,noIn

*AssignmentExpression*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.

5.14.5 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 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 *NextInputElement*^{re}, *NextInputElement*^{div}, and *NextInputElement*^{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 analyses are interleaved.

NOTE The grammar uses *NextInputElement*^{unit} if the previous token was a number, *NextInputElement*^{re} if the previous token was not a number and a / should be interpreted as starting a regular expression, and *NextInputElement*^{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)

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 ▯ can be either re or div.

Lex[*NextInputElement*^{re} ▯ *WhiteSpace InputElement*^{re}] = Lex[*InputElement*^{re}]

Lex[*NextInputElement*^{div} ▯ *WhiteSpace InputElement*^{div}] = Lex[*InputElement*^{div}]

Lex[*NextInputElement*^{unit} ▯ [lookahead▯ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace InputElement*^{div}] = Lex[*InputElement*^{div}]

Lex[*NextInputElement*^{unit} ▯ [lookahead▯ {_}] *IdentifierName*]
  Return a **string** token with string contents LexString[*IdentifierName*].

Lex[*InputElement*▯ ▯ *LineBreaks*] = **lineBreak**

Lex[*InputElement*▯ ▯ *IdentifierOrKeyword*] = Lex[*IdentifierOrKeyword*]

Lex[*InputElement*▯ ▯ *Punctuator*] = Lex[*Punctuator*]

Lex[*InputElement*^{div} ▯ *DivisionPunctuator*] = Lex[*DivisionPunctuator*]

Lex[*InputElement*▯ ▯ *NumericLiteral*] = Lex[*NumericLiteral*]

Lex[*InputElement*▯ ▯ *StringLiteral*] = Lex[*StringLiteral*]

Lex[*InputElement*^{re} ▯ *RegExpLiteral*] = Lex[*RegExpLiteral*]

Lex[*InputElement*▯ ▯ *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 (an integer between 0 and 9).

LexNumber[*NonZeroDigit*]
Return *NonZeroDigit*'s decimal value (an integer between 1 and 9).

LexNumber[*HexDigit*]
Return *HexDigit*'s value (an integer 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 ⟨ can be either single or double.

StringLiteral ⟨
' *StringChars*^{single} '
| " *StringChars*^{double} "

StringChars⟨ ⟨
«empty»
| *StringChars*⟨ *StringChar*⟨
| *StringChars*⟨ *NullEscape* (*NullEscape*: 7.5)

StringChar⟨ ⟨
LiteralStringChar⟨
| \ *StringEscape*

LiteralStringChar^{single} ⟨ *NonTerminator* **except** ' | \ (*NonTerminator*: 7.4)

LiteralStringChar^{double} ⟨ *NonTerminator* **except** " | \

StringEscape ⟨
ControlEscape
| *ZeroEscape*
| *HexEscape*
| *IdentityEscape*

IdentityEscape ⟨ *NonTerminator* **except** _ | *UnicodeAlphanumeric* (*UnicodeAlphanumeric*: 7.5)

ControlEscape ⟨ b | f | n | r | t | v

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

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

Semantics

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

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

LexString[*StringChars*$^{\emptyset}$ :: «empty»] = “”

LexString[*StringChars*$^{\emptyset}$ :: *StringChars*$^{\emptyset}_1$ *StringChar*$^{\emptyset}$]
Return a string consisting of the string LexString[*StringChars*$^{\emptyset}_1$] concatenated with the character LexChar[*StringChar*$^{\emptyset}$].

LexString[*StringChars*$^{\emptyset}$ :: *StringChars*$^{\emptyset}_1$ *NullEscape*] = LexString[*StringChars*$^{\emptyset}_1$]

LexChar[*StringChar*$^{\emptyset}$ :: *LiteralStringChar*$^{\emptyset}$]
Return the character *LiteralStringChar*$^{\emptyset}$.

LexChar[*StringChar*$^{\emptyset}$ :: \ *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 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, a floating-point number, a signed or unsigned 64-bit integer, a character, a string, a namespace, a compound attribute, a class, a method closure, a prototype instance, a class instance, a package object, or the global object. These kinds of objects are described in the subsections below.

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

OBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ LONG ∪ ULONG ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE ∪ PACKAGE ∪ GLOBAL

A GENERALNUMBER is either a floating-point number or a signed or unsigned 64-bit integer:

GENERALNUMBER = FLOAT64 ∪ LONG ∪ ULONG;

A PRIMITIVEOBJECT is either **undefined**, **null**, a Boolean, a floating-point number, a signed or unsigned 64-bit integer, a character, or a string:

PRIMITIVEOBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ LONG ∪ ULONG ∪ CHARACTER ∪ STRING;

A DYNAMICOBJECT is an object that can host dynamic properties:

DYNAMICOBJECT = PROTOTYPE ∪ DYNAMICINSTANCE ∪ GLOBAL;

The semantic domain OBJECTOPT consists of all objects as well as the tag **none** which denotes the absence of an object. **none** is not a value visible to ECMAScript programmers.

OBJECTOPT = OBJECT ∪ {**none**};

The semantic domain OBJECTI consists of all objects as well as the tag **inaccessible** which denotes that a variable's value is not available at this time (for example, a variable whose value is accessible only at run time would hold the value **inaccessible** at compile time). **inaccessible** is not a value visible to ECMAScript programmers.

OBJECTI = OBJECT ∪ {**inaccessible**};

The semantic domain OBJECTIOPT consists of all objects as well as the tags **none** and **inaccessible**:

OBJECTIOPT = OBJECT ∪ {**inaccessible**, **none**};

Some of the variables are in an uninitialised state before first being assigned a value. The semantic domain OBJECTU describes such a variable, which contains either an object or the tag **uninitialised**. **uninitialised** is not a value visible to ECMAScript programmers. The difference between **uninitialised** and **inaccessible** is that a variable holding the value **uninitialised** can be written but not read, while a variable holding the value **inaccessible** can be neither read nor written.

OBJECTU = OBJECT ∪ {**uninitialised**};

The semantic domain BOOLEANOPT consists of the tags **true**, **false**, and **none**:

BOOLEANOPT = BOOLEAN ∪ {**none**};

9.1.1 Undefined

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

UNDEFINED = {**undefined**}

9.1.2 Null

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

NULL = {**null**}

9.1.3 Signed and Unsigned Long Integers

Signed and unsigned long integers are enclosed in tuples in the semantics to distinguish them from mathematically equal floating-point values.

A LONG tuple (see section 5.11) has the field below and represents a signed 64-bit integer.

Field	Contents	Note
value	$\{-2^{63} \ldots 2^{63} - 1\}$	The signed 64-bit integer

A ULONG tuple (see section 5.11) has the field below and represents an unsigned 64-bit integer.

Field	Contents	Note
value	$\{0 \ldots 2^{64} - 1\}$	The unsigned 64-bit integer

~~9.1.3~~9.1.4 Booleans

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

~~9.1.4~~9.1.5 Numbers

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

~~9.1.5~~9.1.6 Strings

The semantic domain 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.

The semantic domain STRINGOPT consists of all strings as well as the tag **none** which denotes the absence of a string. **none** is not a value visible to ECMAScript programmers.

STRINGOPT = STRING ∪ {**none**}

~~9.1.6~~9.1.7 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`

~~9.1.6.1~~9.1.7.1 Qualified Names

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

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

QUALIFIEDNAMEOPT consists of all qualified names as well as **none**:
QUALIFIEDNAMEOPT = QUALIFIEDNAME ∪ {**none**}

MULTINAME is the semantic domain of sets of qualified names. Multinames are used internally in property lookup.
MULTINAME = QUALIFIEDNAME{}

~~9.1.7~~9.1.8 Compound attributes

Compound attribute objects are all values obtained from combining zero or more syntactic attributes (see *****) that are not Booleans or single namespaces. A compound attribute object is represented by a COMPOUNDATTRIBUTE tuple (see section 5.11) with the fields below.

Field	Contents	Note
namespaces	NAMESPACE{}	The set of namespaces contained in this attribute
explicit	BOOLEAN	**true** if the `explicit` attribute has been given
dynamic	BOOLEAN	**true** if the `dynamic` attribute has been given
memberMod	MEMBERMODIFIER	**static**, **constructor**, ~~**operator**,~~ **abstract**, **virtual**, or **final** if one of these attributes has been given; **none** if not. MEMBERMODIFIER = {**none**, **static**, **constructor**, ~~**operator**,~~ **abstract**, **virtual**, **final**}
overrideMod	OVERRIDEMODIFIER	**true**, **false**, or **undefined** if the `override` attribute with one of these arguments was given; **true** if the attribute `override` without arguments was given; **none** if the `override` attribute was not given. OVERRIDEMODIFIER = {**none**, **true**, **false**, **undefined**}
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

ATTRIBUTE consists of all attributes and attribute combinations, including Booleans and single namespaces:
ATTRIBUTE = BOOLEAN ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE

ATTRIBUTEOPTNOTFALSE consists of **none** as well as all attributes and attribute combinations except for **false**:
ATTRIBUTEOPTNOTFALSE = {**none**, **true**} ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE

~~9.1.8~~9.1.9 Classes

Programmer-visible class objects are represented as CLASS records (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable static members defined in this class (see section *****)
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable static members defined in this class
instanceReadBindings	INSTANCEBINDING{}	Map of qualified names to readable instance members defined in this class
instanceWriteBindings	INSTANCEBINDING{}	Map of qualified names to writable instance members defined in this class
instanceInitOrder	INSTANCEVARIABLE[]	List of instance variables defined in this class in the order in which they are initialised
complete	BOOLEAN	**true** after all members of this class have been added to this CLASS record
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**
privateNamespace	NAMESPACE	This class's `private` namespace
dynamic	BOOLEAN	**true** if this class or any of its ancestors was defined with the `dynamic` attribute
allowNull~~primitive~~	BOOLEAN	**true** if ~~**this class was defined with the primitive attribute**~~**null** is considered to be an instance of this class
final	BOOLEAN	**true** if this class cannot be subclassed
call	OBJECT ▯ ARGUMENTLIST ▯ PHASE ▯ OBJECT	A procedure to call (see section 9.5) when this class is used in a call expression
construct	~~OBJECT ▯~~ ARGUMENTLIST ▯ PHASE ▯ OBJECT	A procedure to call (see section 9.5) when this class is used in a `new` expression

CLASSOPT consists of all classes as well as **none**:

CLASSOPT = CLASS ▯ {**none**}

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.

~~9.1.9~~9.1.10 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	INSTANCEMETHOD	The bound method

~~9.1.10~~9.1.11 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; **none** otherwise.
dynamicProperties	DYNAMICPROPERTY{}	A set of this instance's dynamic properties

PROTOTYPEOPT consists of all PROTOTYPE records as well as **none**:

PROTOTYPEOPT = PROTOTYPE ∪ {**none**};

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

~~9.1.11~~9.1.12 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. An instance can also be an ALIASINSTANCE that refers to another instance. This specification uses ALIASINSTANCEs to permit but not require an implementation to share function closures with identical behaviour.

INSTANCE = NONALIASINSTANCE ∪ ALIASINSTANCE;

NONALIASINSTANCE = FIXEDINSTANCE ∪ DYNAMICINSTANCE;

NOTE Instances of some built-in classes are represented as described in sections 9.1.1 through 9.1.11 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	OBJECT ∪ ARGUMENTLIST ∪ ENVIRONMENT ∪ PHASE ∪ OBJECT~~INVOKER~~	A procedure to call when this instance is used in a call expression. The procedure takes an OBJECT (the `this` value), an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
construct	ARGUMENTLIST ∪ ENVIRONMENT ∪ PHASE ∪ OBJECT~~INVOKER~~	A procedure to call when this instance is used in a `new` expression. The procedure takes an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
env	ENVIRONMENT	The environment to pass to the call or construct procedure
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	OBJECT ∪ ARGUMENTLIST ∪ ENVIRONMENT ∪ PHASE ∪	A procedure to call when this instance is used in a call expression. The procedure takes an OBJECT (the `this` value),

	ENVIRONMENT ∪ PHASE ∪ OBJECT~~INVOKER~~	expression. The procedure takes an OBJECT (the `this` value), an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
construct	ARGUMENTLIST ∪ ENVIRONMENT ∪ PHASE ∪ OBJECT~~INVOKER~~	A procedure to call when this instance is used in a `new` expression. The procedure takes an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
env	ENVIRONMENT	The environment to pass to the call or construct procedure
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

ALIASINSTANCE records (see section 5.12) with the fields below represent aliases to existing instances. An ALIASINSTANCE behaves just like its original instance except that it supplies a different environment to the call and construct procedures. In practice, an implementation would likely only use ALIASINSTANCEs if it can prove that supplying the different environment to the call and construct procedures has no visible consequences, so it could optimise out the ALIASINSTANCE altogether.

Field	Contents	Note
original	NONALIASINSTANCE	This original instance being aliased
env	ENVIRONMENT	The environment to pass to the call or construct procedure

~~9.1.11.1~~9.1.12.1 Open Instances

An OPENINSTANCE record (see section 5.12) has the fields below. It is not an instance in itself but creates an instance when instantiated with an environment. OPENINSTANCE records represent functions with variables inherited from their enclosing environments; supplying the environment turns such a function into a callable instance.

Field	Contents	Note
instantiate	ENVIRONMENT ∪ NONALIASINSTANCE	A procedure to call to supply an environment and obtain a fresh instance
cache	NONALIASINSTANCE ∪ {**none**}	Optional cached value of the last instantiation. This cache serves only to precisely specify the closure sharing optimization and would likely not be present in any actual implementation.

~~9.1.11.2~~9.1.12.2 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	INSTANCEVARIABLE	The instance variable whose value this slot carries
value	OBJECTU	This fixed property's current value; **uninitialised** if the fixed property is an uninitialised constant

~~9.1.12~~9.1.13 Packages

Programmer-visible packages are represented as PACKAGE records (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable members defined in this package
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable members defined in this package
internalNamespace	NAMESPACE	This package's `internal` namespace

~~9.1.13~~9.1.14 Global Objects

Programmer-visible global objects are represented as GLOBAL records (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable members defined in this global object
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable members defined in this global object
internalNamespace	NAMESPACE	This global object's `internal` namespace
dynamicProperties	DYNAMICPROPERTY{}	A set of this global object's dynamic properties

9.2 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 References

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

REFERENCE = LEXICALREFERENCE | DOTREFERENCE | BRACKETREFERENCE;

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

OBJORREF = OBJECT | REFERENCE

A LEXICALREFERENCE tuple (see section 5.11) has the fields below and represents an lvalue that refers to a variable with one of a given set of qualified names. LEXICALREFERENCE tuples arise from evaluating identifiers *a* and qualified identifiers *q*`::`*a*.

Field	Contents	Note
env	ENVIRONMENT	The environment in which the reference was created.
variableMultiname	MULTINAME	A nonempty set of qualified names to which this reference can refer
strict~~ext~~	BOOLEAN~~CONTEXT~~	~~The context~~**true** if strict mode was in effect at the point where the reference was created

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 one of a given set of qualified names. DOTREFERENCE tuples arise from evaluating subexpressions such as *a*`.`*b* or *a*`.`*q*`::`*b*.

Field	Contents	Note
base	OBJOPTIONALLIMIT	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.
propertyMultiname	MULTINAME	A nonempty set of qualified names to which this reference can refer (*b* qualified with the namespace *q* or all currently open namespaces in the example 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	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)

~~9.3.1References 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 9.2) before operating on it.~~

~~Some subexpressions evaluate to an OBJORREFOPTIONALLIMIT, which is either an OBJORREF or a LIMITEDOBJORREF:~~

~~OBJORREFOPTIONALLIMIT = OBJORREF | LIMITEDOBJORREF~~

9.4 Function Support

There are ~~four~~ three kinds of functions: normal functions, getters, and setters~~, and operators~~. The FUNCTIONKIND semantic domain encodes the kind:

FUNCTIONKIND = {**normal**, **get**, **set**~~, **operator**~~}

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

Field	Contents	Note
~~requiredPositional~~positional	PARAMETER[]	List of the required positional parameters
optionalPositional	PARAMETER[]	List 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	PARAMETER	{**none**}	The parameter for collecting 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 PARAMETER tuple (see section 5.11) has the fields below and represents the signature of one unnamed parameter.

Field	Contents	Note
localName	QUALIFIEDNAMEOPT~~STRINGOPT~~	Name of the local variable that will hold this parameter's value
type	CLASS	This parameter's type

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

Field	Contents	Note
localName	QUALIFIEDNAMEOPT~~STRINGOPT~~	Name of the local variable that will hold this parameter's value
type	CLASS	This parameter's type
name	STRING	This parameter's external name

9.5 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 of procedures that take an OBJECT (the `this` value), an ARGUMENTLIST, a lexical ENVIRONMENT, and a PHASE (see section 9.8) and produce an OBJECT result:~~

~~INVOKER = OBJECT × ARGUMENTLIST × ENVIRONMENT × PHASE → OBJECT~~

~~9.6Unary 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.~~

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

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

~~9.7Binary 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.~~

~~A BINARYMETHOD tuple (see section) 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~~
~~f~~	~~OBJECT × OBJECT × PHASE → OBJECT~~	~~Procedure that takes the left and right operand values and a PHASE (see section) and returns the operator's result~~

~~9.8~~9.6 Modes of expression evaluation

Expressions can be evaluated in either run mode or compile mode. In run mode all operations are allowed. In compile mode, operations are restricted to those that cannot use or produce side effects, access non-constant variables, or call programmer-defined functions.

The semantic domain PHASE consists of the tags **compile** and **run** representing the two modes of expression evaluation:

PHASE = {**compile**, **run**}

~~9.9~~9.7 Contexts

A CONTEXT tuple (see section 5.11) carries static information about a particular point in the source program and has the fields below.

Field	Contents	Note
strict	BOOLEAN	**true** if strict mode (see *****) is in effect
openNamespaces	NAMESPACE{}	The set of namespaces that are open at this point. The `public` namespace is always a member of this set.

~~9.10~~9.8 Labels

A LABEL is a label that can be used in a `break` or `continue` statement. The label is either a string or the special tag **default**. Strings represent labels named by identifiers, while **default** represents the anonymous label.

LABEL = STRING ∪ {**default**}

A JUMPTARGETS tuple (see section 5.11) describes the sets of labels that are valid destinations for `break` or `continue` statements at a point in the source code. A JUMPTARGETS tuple has the fields below.

Field	Contents	Note
breakTargets	LABEL{}	The set of labels that are valid destinations for a `break` statement
continueTargets	LABEL{}	The set of labels that are valid destinations for a `continue` statement

~~9.11~~9.9 Environments

Environments contain the bindings that are visible from a given point in the source code. An ENVIRONMENT is a list of two or more frames. Each frame corresponds to a scope. More specific frames are listed first—each frame's scope is directly

contained in the following frame's scope. The last frame is always the SYSTEMFRAME. The next-to-last frame is always a PACKAGE or GLOBAL frame.

ENVIRONMENT = FRAME[]

The semantic domain ENVIRONMENTI consists of all environments as well as the tag **inaccessible** which denotes that an environment is not available at this time:

ENVIRONMENTI = ENVIRONMENT ∪ {**inaccessible**};

~~9.11.1~~9.9.1 Frames

A frame contains bindings defined at a particular scope in a program. A frame is either the top-level system frame, a global object, a package, a function parameter frame, a class, or a block frame:

FRAME = SYSTEMFRAME ∪ GLOBAL ∪ PACKAGE ∪ PARAMETERFRAME~~FUNCTIONFRAME~~ ∪ CLASS ∪ BLOCKFRAME;

Some frames can be marked either **singular** or **plural**. A **singular** frame contains the current values of variables and other definitions. A **plural** frame is a template for making **singular** frames — a **plural** frame contains placeholders for mutable variables and definitions as well as the actual values of compile-time constant definitions. The static analysis done by Validate generates **singular** frames for the system frame, global object, and any blocks, classes, or packages directly contained inside another **singular** frame; all other frames are **plural** during static analysis and are instantiated to make **singular** frames by Eval.

The system frame, global objects, packages, and classes are always **singular**. Function and block frames can be either **singular** or **plural**.

PLURALITY is the semantic domain of the two tags **singular** and **plural**:

PLURALITY = {**singular**, **plural**}

~~9.11.1.1~~9.9.1.1 System Frame

The top-level frame containing predefined constants, functions, and classes is represented as a SYSTEMFRAME record (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable definitions in this frame
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable definitions in this frame

~~9.11.1.2~~9.9.1.2 Function Parameter Frames

Frames holding bindings for invoked functions are represented as PARAMETERFRAME~~FUNCTIONFRAME~~ records (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable definitions in this function
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable definitions in this function
plurality	PLURALITY	See section 9.9.1
this	OBJECTIOPT	The value of `this`; **none** if this function doesn't define `this`; **inaccessible** if this function defines `this` but the value is not available because this function hasn't been called yet
prototype	BOOLEAN	**true** if this function is not an instance method but defines `this` anyway
signature	SIGNATURE	This function's signature

~~9.11.1.3~~9.9.1.3 Block Frames

Frames holding bindings for blocks are represented as BLOCKFRAME records (see section 5.12) with the fields below.

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable definitions in this block
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable definitions in this block
plurality	PLURALITY	See section 9.9.1

~~9.11.2~~9.9.2 Static Bindings

A STATICBINDING tuple (see section 5.11) has the fields below and describes the member to which one qualified name is bound in a frame. Multiple qualified names may be bound to the same member in a frame, but a qualified name may not be bound to multiple members in a frame (except when one binding is for reading only and the other binding is for writing only).

Field	Contents	Note
qname	QUALIFIEDNAME	The qualified name bound by this binding
content	STATICMEMBER	The member to which this qualified name was bound
explicit	BOOLEAN	**true** if this binding should not be imported into the global scope by an `import` statement

A static member is either **forbidden**, a variable, a hoisted variable, a constructor method, ~~or an accessor~~a getter, or a setter:

STATICMEMBER = {**forbidden**} ∪ VARIABLE ∪ HOISTEDVAR ∪ CONSTRUCTORMETHOD ∪ GETTER ∪ SETTER~~ACCESSOR~~;

STATICMEMBEROPT = STATICMEMBER ∪ {**none**};

A **forbidden** static member is one that must not be accessed because there exists a definition for the same qualified name in a more local block.

A VARIABLE record (see section 5.12) has the fields below and describes one variable or constant definition.

Field	Contents	Note
type	VARIABLETYPE	Type of values that may be stored in this variable (see below)
value	VARIABLEVALUE	This variable's current value; **future** if the variable has not been declared yet; **uninitialised** if the variable must be written before it can be read
immutable	BOOLEAN	**true** if this variable's value may not be changed once set

A variable's type can be either a class, **inaccessible**, or a ~~future type~~semantic procedure that takes no parameters and will compute a class on demand; such procedures are used instead of CLASSes for types of variables in situations where the type expression can contain forward references and shouldn't be evaluated until it is needed.~~:~~

VARIABLETYPE = CLASS ∪ {**inaccessible**} ∪ () → CLASS~~FUTURETYPE~~

~~A FUTURETYPE record (see section) has the field below. It is a wrapper for a procedure that produces a type. FUTURETYPEs are used for the types of variables instead of CLASSes in situations where the type expression can contain forward references and shouldn't be evaluated until it is needed.~~

~~Field~~	~~Contents~~	~~Note~~
~~evalType~~	~~() → CLASS~~	~~A procedure to call to get the type~~

A variable's value can be either an object, **inaccessible** (used when the variable has not been declared yet), **uninitialised** (used when the variable must be written before it can be read), an open (unclosed) function (compile time only), or a semantic procedure (compile time only) that takes no parameters and will compute an object on demand~~future value (compile time only)~~; such procedures are used instead of OBJECTs for values of compile-time constants in situations where the value expression can contain forward references and shouldn't be evaluated until it is needed~~:~~.

VARIABLEVALUE = OBJECT ∪ {**inaccessible**, **uninitialised**} ∪ OPENINSTANCE ∪ () → OBJECT~~FUTUREVALUE~~;

~~A FUTUREVALUE record (see section) has the field below. It is a wrapper for a procedure that produces a type. FUTUREVALUES are used for the values of compile-time constants instead of OBJECTS in situations where the value expression can contain forward references and shouldn't be evaluated until it is needed.~~

~~Field~~	~~Contents~~	~~Note~~
~~evalValue~~	~~() → OBJECT~~	~~A procedure to call to get the value~~

A HOISTEDVAR record (see section 5.12) has the fields below and describes one hoisted variable.

Field	Contents	Note
value	OBJECT ∪ OPENINSTANCE	This variable's current value; may be an open (unclosed) function at compile time
hasFunctionInitialiser	BOOLEAN	**true** if this variable was created by a `function` statement

A CONSTRUCTORMETHOD record (see section 5.12) has the field below and describes one constructor definition.

Field	Contents	Note
code	INSTANCE	This constructor itself (a callable object)

A GETTER record (see section 5.12) has the fields below and describes one static getter definition.

Field	Contents	Note
type	CLASS	The type of the value read from this getter
call	ENVIRONMENT × PHASE → OBJECT	A procedure to call to read the value, passing it the environment from the env field below and the current mode of expression evaluation
env	ENVIRONMENTI	The environment bound to this getter

A SETTER record (see section 5.12) has the fields below and describes one static setter definition.

Field	Contents	Note
type	CLASS	The type of the value written by this setter
call	OBJECT × ENVIRONMENT × PHASE → ()	A procedure to call to write the value, passing it the new value, the environment from the env field below, and the current mode of expression evaluation
env	ENVIRONMENTI	The environment bound to this setter

~~9.11.3~~9.9.3 Instance Bindings

An INSTANCEBINDING tuple (see section 5.11) has the fields below and describes the binding of one qualified name to an instance member of a class. Multiple qualified names may be bound to the same instance member in a class, but a qualified name may not be bound to multiple instance members in a class (except when one binding is for reading only and the other binding is for writing only).

Field	Contents	Note
qname	QUALIFIEDNAME	The qualified name bound by this binding
content	INSTANCEMEMBER	The member to which this qualified name was bound

An instance member is either an instance variable, an instance method, or an instance accessor:

INSTANCEMEMBER = INSTANCEVARIABLE ∪ INSTANCEMETHOD ∪ INSTANCEGETTER ∪ INSTANCESETTER~~INSTANCEACCESSOR~~;

INSTANCEMEMBEROPT = INSTANCEMEMBER ∪ {**none**};

An INSTANCEVARIABLE record (see section 5.12) has the fields below and describes one instance variable or constant definition.

Field	Contents	Note
type	CLASS	Type of values that may be stored in this variable
evalInitialValue	() → OBJECTOPT	A function that computes this variable's initial value
immutable	BOOLEAN	**true** if this variable's value may not be changed once set
final	BOOLEAN	**true** if this member may not be overridden in subclasses

An INSTANCEMETHOD record (see section 5.12) has the fields below and describes one instance method definition.

Field	Contents	Note
code	INSTANCE → {**abstract**}	This method itself (a callable object); **abstract** if this method is abstract
signature	SIGNATURE	This method's signature
final	BOOLEAN	**true** if this member may not be overridden in subclasses

An INSTANCEGETTER record (see section 5.12) has the fields below and describes one instance getter definition.

Field	Contents	Note
type	CLASS	The type of the value read from this getter
call	OBJECT → ENVIRONMENT → PHASE → OBJECT	A procedure to call to read the value, passing it the `this` value, the environment from the `env` field below, and the current mode of expression evaluation
env	ENVIRONMENTI	The environment bound to this getter
final	BOOLEAN	**true** if this member may not be overridden in subclasses

An INSTANCESETTER record (see section 5.12) has the fields below and describes one instance setter definition.

Field	Contents	Note
type	CLASS	The type of the value written by this setter
call	OBJECT → OBJECT → ENVIRONMENT → PHASE → ()	A procedure to call to write the value, passing it the new value, the `this` value, the environment from the `env` field below, and the current mode of expression evaluation
env	ENVIRONMENTI	The environment bound to this setter
final	BOOLEAN	**true** if this member may not be overridden in subclasses

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

unsignedWrap32(*i*) returns *i* converted to a value between 0 and 2^{32}–1 inclusive, wrapping around modulo 2^{32} if necessary.

```
proc unsignedWrap32(i: INTEGER): {0 ... 2^32 – 1}
    return bitwiseAnd(i, 0xFFFFFFFF)
end proc;
```

signedWrap32(*i*) returns *i* converted to a value between -2^{31} and $2^{31}-1$ inclusive, wrapping around modulo 2^{32} if necessary.

```
proc signedWrap32(i: INTEGER): {–2^31 ... 2^31 – 1}
    j: INTEGER ← bitwiseAnd(i, 0xFFFFFFFF);
    if j ≥ 2^31 then j ← j – 2^32 end if;
    return j
end proc;
```

unsignedWrap64(*i*) returns *i* converted to a value between 0 and $2^{64}-1$ inclusive, wrapping around modulo 2^{64} if necessary.

```
proc unsignedWrap64(i: INTEGER): {0 ... 2^64 – 1}
    return bitwiseAnd(i, 0xFFFFFFFFFFFFFFFF)
end proc;
```

signedWrap64(*i*) returns *i* converted to a value between -2^{63} and $2^{63}-1$ inclusive, wrapping around modulo 2^{64} if necessary.

```
proc signedWrap64(i: INTEGER): {–2^63 ... 2^63 – 1}
    j: INTEGER ← bitwiseAnd(i, 0xFFFFFFFFFFFFFFFF);
    if j ≥ 2^63 then j ← j – 2^64 end if;
    return j
end proc;

proc truncateToInteger(x: GENERALNUMBER): INTEGER
    case x of
        {+∞, –∞, NaN} do return 0;
        FINITEFLOAT64 do return truncateFiniteFloat64(x);
        LONG ∪ ULONG do return x.value
    end case
end proc;

proc checkInteger(x: GENERALNUMBER): INTEGER
    if x = NaN then throw badValueError end if;
    if x ∈ {+∞, –∞} then throw rangeError end if;
    r: RATIONAL ← toRational(x);
    if r ∉ INTEGER then throw badValueError end if;
    return r
end proc;

proc checkLong(i: INTEGER): LONG
    if –2^63 ≤ i ≤ 2^63 – 1 then return LONG{value: i} else throw rangeError end if
end proc;

proc checkULong(i: INTEGER): ULONG
    if 0 ≤ i ≤ 2^64 – 1 then return ULONG{value: i} else throw rangeError end if
end proc;

proc toRational(x: FINITEFLOAT64 ∪ LONG ∪ ULONG): RATIONAL
    case x of
        {+zero, –zero} do return 0;
        NONZEROFINITEFLOAT64 do return x;
        LONG ∪ ULONG do return x.value
    end case
end proc;
```

```
proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
    if x ∈ FLOAT64 and y ∈ FLOAT64 then return float64Compare(x, y)
    elsif x = NaN or y = NaN then return unordered
    elsif x = +∞ or y = –∞ then return greater
    elsif x = –∞ or y = +∞ then return less
    else return rationalCompare(toRational(x), toRational(y))
    end if
end proc;

proc float64ToString(x: FLOAT64): STRING
    ????
end proc;
```

10.2 Object Utilities

```
proc resolveAlias(o: INSTANCE): NONALIASINSTANCE
    case o of
        NONALIASINSTANCE do return o;
        ALIASINSTANCE do return o.original
    end case
end proc;
```

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;
        LONG do return longClass;
        ULONG do return uLongClass;
        CHARACTER do return characterClass;
        STRING do return stringClass;
        NAMESPACE do return namespaceClass;
        COMPOUNDATTRIBUTE do return attributeClass;
        CLASS do return classClass;
        METHODCLOSURE do return functionClass;
        PROTOTYPE do return prototypeClass;
        INSTANCE do return resolveAlias(o).type;
        PACKAGE ∪ GLOBAL do return packageClass
    end case
end proc;
```

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
    return isAncestor(c, objectType(o))
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);
    return isAncestor(c, t) or (o = null and c.allowNull)
end proc;
```

10.2.3 *toBoolean*

toBoolean(*o*, *phase*) coerces an object *o* to a Boolean. If *phase* is **compile**, only compile-time conversions are permitted.

```
proc toBoolean(o: OBJECT, phase: PHASE): BOOLEAN
    case o of
        UNDEFINED ∪ NULL do return false;
        BOOLEAN do return o;
        FLOAT64 do return o ∈ {+zero, –zero, NaN};
        LONG ∪ ULONG do return o.value ≠ 0;
        STRING do return o ≠ “”;
        CHARACTER ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE ∪
            PACKAGE ∪ GLOBAL do
            return true
    end case
end proc;
```

10.2.4 *toGeneralNumber*

toGeneralNumber(*o*, *phase*) coerces an object *o* to a GENERALNUMBER. If *phase* is **compile**, only compile-time conversions are permitted.

```
proc toGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
    case o of
        UNDEFINED do return NaN;
        NULL ∪ {false} do return +zero;
        {true} do return 1.0;
        GENERALNUMBER do return o;
        CHARACTER ∪ STRING do ????;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PACKAGE ∪ GLOBAL do
            throw badValueError;
        PROTOTYPE ∪ INSTANCE do ????
    end case
end proc;
```

10.2.5 *toString*

toString(*o*, *phase*) coerces an object *o* to a string. If *phase* is **compile**, only compile-time conversions are permitted.

```
proc toString(o: OBJECT, phase: PHASE): STRING
    case o of
        UNDEFINED do return "undefined";
        NULL do return "null";
        {false} do return "false";
        {true} do return "true";
        FLOAT64 do return float64ToString(o);
        LONG ∪ ULONG do return integerToString(o.value);
        CHARACTER do return [o];
        STRING do return o;
        NAMESPACE do ????;
        COMPOUNDATTRIBUTE do ????;
        CLASS do ????;
        METHODCLOSURE do ????;
        PROTOTYPE ∪ INSTANCE do ????;
        PACKAGE ∪ GLOBAL do ????
    end case
end proc;
```

integerToString(*i*) converts an integer *i* to a string of one or more decimal digits. If *i* is negative, the string is preceded by a minus sign.

```
proc integerToString(i: INTEGER): STRING
    if i < 0 then return ['-'] ⊕ integerToString(–i) end if;
    q: INTEGER ← ⌊i/10⌋;
    r: INTEGER ← i – q×10;
    c: CHARACTER ← codeToCharacter(r + characterToCode('0'));
    if q = 0 then return [c] else return integerToString(q) ⊕ [c] end if
end proc;
```

10.2.6 *toPrimitive*

```
proc toPrimitive(o: OBJECT, hint: OBJECT, phase: PHASE): PRIMITIVEOBJECT
    case o of
        PRIMITIVEOBJECT do return o;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE ∪ PACKAGE ∪
            GLOBAL do
            return toString(o, phase)
    end case
end proc;
```

10.2.7 *assignmentConversion*

```
proc assignmentConversion(o: OBJECT, type: CLASS): OBJECT
    if relaxedHasType(o, type) then return o end if;
    ????
end proc;
```

10.2.8 Attributes

combineAttributes(*a*, *b*) returns the attribute that results from concatenating the attributes *a* and *b*.

```
proc combineAttributes(a: AttributeOptNotFalse, b: Attribute): Attribute
    if b = false then return false
    elsif a ∈ {none, true} then return b
    elsif b = true then return a
    elsif a ∈ Namespace then
        if a = b then return a
        elsif b ∈ Namespace then
            return CompoundAttribute[namespaces: {a, b}, explicit: false, dynamic: false, memberMod: none,
                    overrideMod: none, prototype: false, unused: false]
        else return CompoundAttribute[namespaces: b.namespaces ∪ {a}, other fields from b]
        end if
    elsif b ∈ Namespace then
        return CompoundAttribute[namespaces: a.namespaces ∪ {b}, other fields from a]
    else
        Both a and b are compound attributes. Ensure that they have no conflicting contents.
        if (a.memberMod ≠ none and b.memberMod ≠ none and a.memberMod ≠ b.memberMod) or
                (a.overrideMod ≠ none and b.overrideMod ≠ none and a.overrideMod ≠ b.overrideMod) then
            throw badValueError
        else
            return CompoundAttribute[namespaces: a.namespaces ∪ b.namespaces,
                    explicit: a.explicit or b.explicit, dynamic: a.dynamic or b.dynamic,
                    memberMod: a.memberMod ≠ none ? a.memberMod : b.memberMod,
                    overrideMod: a.overrideMod ≠ none ? a.overrideMod : b.overrideMod,
                    prototype: a.prototype or b.prototype, unused: a.unused or b.unused]
        end if
    end if
end proc;
```

toCompoundAttribute(*a*) returns *a* converted to a CompoundAttribute even if it was a simple namespace, **true**, or **none**.

```
proc toCompoundAttribute(a: AttributeOptNotFalse): CompoundAttribute
    case a of
        {none, true} do
            return CompoundAttribute[namespaces: {}, explicit: false, dynamic: false, memberMod: none,
                    overrideMod: none, prototype: false, unused: false]
        Namespace do
            return CompoundAttribute[namespaces: {a}, explicit: false, dynamic: false, memberMod: none,
                    overrideMod: none, prototype: false, unused: false]
        CompoundAttribute do return a
    end case
end proc;
```

10.3 References

If *r* is an Object, *readReference*(*r*, *phase*) returns it unchanged. If *r* is a Reference, this function reads *r* and returns the result. If *phase* is **compile**, only compile-time expressions can be evaluated in the process of reading *r*.

```
proc readReference(r: ObjOrRef, phase: Phase): Object
    case r of
        Object do return r;
        LexicalReference do return lexicalRead(r.env, r.variableMultiname, phase);
        DotReference do
            result: ObjectOpt ← readProperty(r.base, r.propertyMultiname, propertyLookup, phase);
            if result ≠ none then return result else throw propertyAccessError end if;
        BracketReference do return bracketRead(r.base, r.args, phase)
    end case
end proc;
```

```
proc bracketRead(a: ObjOptionalLimit, args: ArgumentList, phase: Phase): Object
    if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
    name: String ← toString(args.positional[0], phase);
    result: ObjectOpt ← readProperty(a, {QualifiedName[namespace: publicNamespace, id: name]},
            propertyLookup, phase);
    if result ≠ none then return result else throw propertyAccessError end if
end proc;
```

If *r* is a reference, *writeReference*(*r*, *newValue*) writes *newValue* into *r*. An error occurs if *r* is not a reference. *r*'s limit, if any, is ignored. *writeReference* is never called from a compile-time expression.

```
proc writeReference(r: ObjOrRef, newValue: Object, phase: {run})
    result: {none, ok};
    case r of
        Object do throw referenceError;
        LexicalReference do
            lexicalWrite(r.env, r.variableMultiname, newValue, not r.strict, phase);
            return;
        DotReference do
            result ← writeProperty(r.base, r.propertyMultiname, propertyLookup, true, newValue, phase);
        BracketReference do result ← bracketWrite(r.base, r.args, newValue, phase)
    end case;
    if result = none then throw propertyAccessError end if
end proc;

proc bracketWrite(a: ObjOptionalLimit, args: ArgumentList, newValue: Object, phase: Phase): {none, ok}
    if phase = compile then throw compileExpressionError end if;
    if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
    name: String ← toString(args.positional[0], phase);
    return writeProperty(a, {QualifiedName[namespace: publicNamespace, id: name]}, propertyLookup, true,
            newValue, phase)
end proc;
```

If *r* is a Reference, *deleteReference*(*r*) deletes it. If *r* is an Object, this function signals an error in strict mode or returns True in non-strict mode. *deleteReference* is never called from a compile-time expression.

```
proc deleteReference(r: ObjOrRef, strict: Boolean, phase: {run}): Boolean
    result: BooleanOpt;
    case r of
        Object do if strict then throw referenceError else return true end if;
        LexicalReference do return lexicalDelete(r.env, r.variableMultiname, phase);
        DotReference do
            result ← deleteProperty(r.base, r.propertyMultiname, propertyLookup, phase);
        BracketReference do result ← bracketDelete(r.base, r.args, phase)
    end case;
    if result ≠ none then return result else return true end if
end proc;

proc bracketDelete(a: ObjOptionalLimit, args: ArgumentList, phase: Phase): BooleanOpt
    if phase = compile then throw compileExpressionError end if;
    if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
    name: String ← toString(args.positional[0], phase);
    return deleteProperty(a, {QualifiedName[namespace: publicNamespace, id: name]}, propertyLookup, phase)
end proc;
```

10.4 Slots

proc *findSlot*(*o*: OBJECT, *id*: INSTANCEVARIABLE): SLOT
  o must be an INSTANCE;
  matchingSlots: SLOT{} ← {*s* | ∈*s* ∈ *resolveAlias*(*o*).slots **such that** *s*.id = *id*};
  return the one element of *matchingSlots*
end proc;

10.5 Environments

If *env* is from within a class's body, *getEnclosingClass*(*env*) returns the innermost such class; otherwise, it returns **none**.

proc *getEnclosingClass*(*env*: ENVIRONMENT): CLASSOPT
  if some *c* ∈ *env* **satisfies** *c* ∈ CLASS **then**
    Let *c* be the first element of *env* that is a CLASS.
    return *c*
  end if;
  return **none**
end proc;

getRegionalEnvironment(*env*) returns all frames in *env* up to and including the first regional frame. A regional frame is either any frame other than a local block frame or a local block frame whose immediate enclosing frame is a class.

proc *getRegionalEnvironment*(*env*: ENVIRONMENT): FRAME[]
  i: INTEGER ← 0;
  while *env*[*i*] ∈ BLOCKFRAME **do** *i* ← *i* + 1 **end while**;
  if *i* ≠ 0 **and** *env*[*i*] ∈ CLASS **then** *i* ← *i* – 1 **end if**;
  return *env*[0 ... *i*]
end proc;

getRegionalFrame(*env*) returns the most specific regional frame in *env*.

proc *getRegionalFrame*(*env*: ENVIRONMENT): FRAME
  regionalEnv: FRAME[] ← *getRegionalEnvironment*(*env*);
  return *regionalEnv*[|*regionalEnv*| – 1]
end proc;

proc *getPackageOrGlobalFrame*(*env*: ENVIRONMENT): PACKAGE ∪ GLOBAL
  g: FRAME ← *env*[|*env*| – 2];
  The penultimate frame *g* is always a PACKAGE or GLOBAL frame.
  return *g*
end proc;

10.5.1 Access Utilities

tag **read**;

tag **write**;

tag **readWrite**;

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

staticBindingsWithAccess(*f*, *access*) returns the set of static bindings in frame *f* which are used for reading, writing, or either, as selected by *access*.

```
proc staticBindingsWithAccess(f: FRAME, access: ACCESS): STATICBINDING{}
    case access of
        {read} do return f.staticReadBindings;
        {write} do return f.staticWriteBindings;
        {readWrite} do return f.staticReadBindings ∪ f.staticWriteBindings
    end case
end proc;
```

instanceBindingsWithAccess(*c*, *access*) returns the set of instance bindings in class *c* which are used for reading, writing, or either, as selected by *access*.

```
proc instanceBindingsWithAccess(c: CLASS, access: ACCESS): INSTANCEBINDING{}
    case access of
        {read} do return c.instanceReadBindings;
        {write} do return c.instanceWriteBindings;
        {readWrite} do return c.instanceReadBindings ∪ c.instanceWriteBindings
    end case
end proc;
```

addStaticBindings(*f*, *access*, *newBindings*) adds *newBindings* to the set of readable, writable, or both (as selected by *access*) static bindings in frame *f*.

```
proc addStaticBindings(f: FRAME, access: ACCESS, newBindings: STATICBINDING{})
    if access ∈ {read, readWrite} then
        f.staticReadBindings ← f.staticReadBindings ∪ newBindings
    end if;
    if access ∈ {write, readWrite} then
        f.staticWriteBindings ← f.staticWriteBindings ∪ newBindings
    end if
end proc;
```

10.5.2 Adding Static Definitions

```
proc defineStaticMember(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE{},
        overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: STATICMEMBER): MULTINAME
    localFrame: FRAME ← env[0];
    if overrideMod ≠ none or (explicit and localFrame ∉ PACKAGE) then
        throw definitionError
    end if;
    namespaces2: NAMESPACE{} ← namespaces;
    if namespaces2 = {} then namespaces2 ← {publicNamespace} end if;
    multiname: MULTINAME ← {QUALIFIEDNAME(namespace: ns, id: id) | ns ∈ namespaces2};
    regionalEnv: FRAME[] ← getRegionalEnvironment(env);
    regionalFrame: FRAME ← regionalEnv[|regionalEnv| – 1];
    if some b ∈ staticBindingsWithAccess(localFrame, access) satisfies b.qname ∈ multiname then
        throw definitionError
    end if;
    for each frame ∈ regionalEnv[1 ...] do
        if some b ∈ staticBindingsWithAccess(frame, access) satisfies
                b.qname ∈ multiname and b.content ≠ forbidden then
            throw definitionError
        end if
    end for each;
    if regionalFrame ∈ GLOBAL and (some dp ∈ regionalFrame.dynamicProperties satisfies
            QUALIFIEDNAME(namespace: publicNamespace, id: dp.name) ∈ multiname) then
        throw definitionError
    end if;
    newBindings: STATICBINDING{} ← {STATICBINDING(qname: qname, content: m, explicit: explicit) |
            qname ∈ multiname};
    addStaticBindings(localFrame, access, newBindings);
    Mark the bindings of multiname as forbidden in all non-innermost frames in the current region if they haven't been
            marked as such already.
    newForbiddenBindings: STATICBINDING{} ← {STATICBINDING(qname: qname, content: forbidden, explicit: true) |
            qname ∈ multiname};
    for each frame ∈ regionalEnv[1 ...] do
        addStaticBindings(frame, access, newForbiddenBindings)
    end for each;
    return multiname
end proc;
```

```
proc defineHoistedVar(env: ENVIRONMENT, id: STRING)
    qname: QUALIFIEDNAME ← QUALIFIEDNAME⟦namespace: publicNamespace, id: id⟧;
    regionalEnv: FRAME[] ← getRegionalEnvironment(env);
    regionalFrame: FRAME ← regionalEnv[|regionalEnv| – 1];
    env is either the GLOBAL frame or a PARAMETERFRAME because hoisting only occurs into global or function scope.
    existingBindings: STATICBINDING{} ← {b | ∃b ∈ staticBindingsWithAccess(regionalFrame, readWrite) such that
            b.qname = qname};
    if existingBindings = {} then
        case regionalFrame of
            GLOBAL do
                if some dp ∈ regionalFrame.dynamicProperties satisfies dp.name = id then
                    throw definitionError
                end if;
            PARAMETERFRAME do
                if |regionalEnv| ≥ 2 then
                    regionalFrame ← regionalEnv[|regionalEnv| – 2];
                    existingBindings ← {b | ∃b ∈ staticBindingsWithAccess(regionalFrame, readWrite) such that
                            b.qname = qname}
                end if
        end case
    end if;
    if existingBindings = {} then
        v: HOISTEDVAR ← new HOISTEDVAR⟦value: undefined, hasFunctionInitialiser: false⟧;
        addStaticBindings(regionalFrame, readWrite, {STATICBINDING⟦qname: qname, content: v, explicit: false⟧})
    elsif some b ∈ existingBindings satisfies b.content ∈ HOISTEDVAR then
        throw definitionError
    else
        A hoisted binding of the same var already exists, so there is no need to create another one.
    end if
end proc;
```

10.5.3 Adding Instance Definitions

```
tuple OVERRIDESTATUSPAIR
    readStatus: OVERRIDESTATUS,
    writeStatus: OVERRIDESTATUS
end tuple;

tag potentialConflict;

OVERRIDDENMEMBER = INSTANCEMEMBER ∪ {none, potentialConflict};

tuple OVERRIDESTATUS
    overriddenMember: OVERRIDDENMEMBER,
    multiname: MULTINAME
end tuple;
```

```
proc searchForOverrides(c: CLASS, id: STRING, namespaces: NAMESPACE{}, access: {read, write}): OVERRIDESTATUS
    multiname: MULTINAME ← {};
    overriddenMember: INSTANCEMEMBEROPT ← none;
    s: CLASSOPT ← c.super;
    for each ns ∈ namespaces do
        qname: QUALIFIEDNAME ← QUALIFIEDNAME[namespace: ns, id: id]
        m: INSTANCEMEMBEROPT ← findInstanceMember(s, qname, access);
        if m ≠ none then
            multiname ← multiname ∪ {qname};
            if overriddenMember = none then overriddenMember ← m
            elsif overriddenMember ≠ m then throw definitionError
            end if
        end if
    end for each;
    return OVERRIDESTATUS[overriddenMember: overriddenMember, multiname: multiname]
end proc;

proc resolveOverrides(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE{}, access: {read, write},
        expectMethod: BOOLEAN): OVERRIDESTATUS
    os: OVERRIDESTATUS;
    if namespaces = {} then
        os ← searchForOverrides(c, id, cxt.openNamespaces, access);
        if os.overriddenMember = none then
            os ← OVERRIDESTATUS[overriddenMember: none,
                    multiname: {QUALIFIEDNAME[namespace: publicNamespace, id: id]}]
        end if
    else
        definedMultiname: MULTINAME ← {QUALIFIEDNAME[namespace: ns, id: id] | ns ∈ namespaces};
        os2: OVERRIDESTATUS ← searchForOverrides(c, id, namespaces, access);
        if os2.overriddenMember = none then
            os3: OVERRIDESTATUS ← searchForOverrides(c, id, cxt.openNamespaces – namespaces, access);
            if os3.overriddenMember = none then
                os ← OVERRIDESTATUS[overriddenMember: none, multiname: definedMultiname]
            else
                os ← OVERRIDESTATUS[overriddenMember: potentialConflict, multiname: definedMultiname]
            end if
        else
            os ← OVERRIDESTATUS[overriddenMember: os2.overriddenMember,
                    multiname: os2.multiname ∪ definedMultiname]
        end if
    end if;
    if some b ∈ instanceBindingsWithAccess(c, access) satisfies b.qname ∈ os.multiname then
        throw definitionError
    end if;
    if expectMethod then
        if os.overriddenMember ∉ {none, potentialConflict} ∪ INSTANCEMETHOD then
            throw definitionError
        end if
    else
        if os.overriddenMember ∉ {none, potentialConflict} ∪ INSTANCEVARIABLE ∪ INSTANCEGETTER ∪
                INSTANCESETTER then
            throw definitionError
        end if
    end if;
    return os
end proc;
```

```
proc defineInstanceMember(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE{},
      overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: INSTANCEMEMBER):
      OVERRIDESTATUSPAIR
   if explicit then throw definitionError end if;
   expectMethod: BOOLEAN ← m ∈ INSTANCEMETHOD;
   readStatus: OVERRIDESTATUS ← access ∈ {read, readWrite} ?
         resolveOverrides(c, cxt, id, namespaces, read, expectMethod) :
         OVERRIDESTATUS[overriddenMember: none, multiname: {}];
   writeStatus: OVERRIDESTATUS ← access ∈ {write, readWrite} ?
         resolveOverrides(c, cxt, id, namespaces, write, expectMethod) :
         OVERRIDESTATUS[overriddenMember: none, multiname: {}];
   if readStatus.overriddenMember ∈ INSTANCEMEMBER or
         writeStatus.overriddenMember ∈ INSTANCEMEMBER then
      if overrideMod ∉ {true, undefined} then throw definitionError end if
   elsif readStatus.overriddenMember = potentialConflict or
         writeStatus.overriddenMember = potentialConflict then
      if overrideMod ∉ {false, undefined} then throw definitionError end if
   else if overrideMod ∉ {none, false, undefined} then throw definitionError end if
   end if;
   newReadBindings: INSTANCEBINDING{} ←
         {INSTANCEBINDING[qname: qname, content: m]| ∃qname ∈ readStatus.multiname};
   c.instanceReadBindings ← c.instanceReadBindings ∪ newReadBindings;
   newWriteBindings: INSTANCEBINDING{} ←
         {INSTANCEBINDING[qname: qname, content: m]| ∃qname ∈ writeStatus.multiname};
   c.instanceWriteBindings ← c.instanceWriteBindings ∪ newWriteBindings;
   return OVERRIDESTATUSPAIR[readStatus: readStatus, writeStatus: writeStatus]
end proc;
```

10.5.4 Instantiation

```
proc instantiateOpenInstance(oi: OPENINSTANCE, env: ENVIRONMENT): INSTANCE
   cache: FIXEDINSTANCE ∪ DYNAMICINSTANCE ∪ {none} ← oi.cache;
   if cache = none then
      i: NONALIASINSTANCE ← oi.instantiate(env);
      reuse: BOOLEAN;
      At the implementation's discretion, either reuse ← true, or reuse ← false. An implementation may make different
            choices at different times. The intent here is to allow implementations the freedom to reuse a closure object
            rather than create a new closure each time a particular OPENINSTANCE is instantiated if the implementation
            notices that the resulting closures would be behaviorally indistinguishable from each other.
      if reuse then oi.cache ← i end if;
      return i
   else return new ALIASINSTANCE[[original: cache, env: env]]
   end if
end proc;
```

```
proc instantiateMember(m: STATICMEMBER, env: ENVIRONMENT): STATICMEMBER
    case m of
        {forbidden} do return m;
        VARIABLE do
            value: VARIABLEVALUE ∪ m.value;
            if value ∈ OPENINSTANCE then value ← instantiateOpenInstance(value, env)
            end if;
            return new VARIABLE[type: m.type, value: value, immutable: m.immutable]
        HOISTEDVAR do
            value: OBJECT ∪ OPENINSTANCE ∪ m.value;
            if value ∈ OPENINSTANCE then value ← instantiateOpenInstance(value, env)
            end if;
            return new HOISTEDVAR[value: value, hasFunctionInitialiser: m.hasFunctionInitialiser]
        CONSTRUCTORMETHOD do return m;
        GETTER do
            case m.env of
                ENVIRONMENT do return m;
                {inaccessible} do return new GETTER[type: m.type, call: m.call, env: env]
            end case;
        SETTER do
            case m.env of
                ENVIRONMENT do return m;
                {inaccessible} do return new SETTER[type: m.type, call: m.call, env: env]
            end case
    end case
end proc;

tuple MEMBERINSTANTIATION
    pluralMember: STATICMEMBER,
    singularMember: STATICMEMBER
end tuple;

proc instantiateFrame(pluralFrame: PARAMETERFRAME ∪ BLOCKFRAME,
        singularFrame: PARAMETERFRAME ∪ BLOCKFRAME, env: ENVIRONMENT)
    pluralMembers: STATICMEMBER{} ← {b.content |
            ∃b ∈ pluralFrame.staticReadBindings ∪ pluralFrame.staticWriteBindings};
    memberInstantiations: MEMBERINSTANTIATION{} ←
            {MEMBERINSTANTIATION[pluralMember: m, singularMember: instantiateMember(m, env)] |
            ∃m ∈ pluralMembers};
    proc instantiateBinding(b: STATICBINDING): STATICBINDING
        mi: MEMBERINSTANTIATION ← the one element mi ∈ memberInstantiations that satisfies mi.pluralMember =
                b.content;
        return STATICBINDING[qname: b.qname, content: mi.singularMember, explicit: b.explicit]
    end proc;
    singularFrame.staticReadBindings ← {instantiateBinding(b) | ∃b ∈ pluralFrame.staticReadBindings};
    singularFrame.staticWriteBindings ← {instantiateBinding(b) | ∃b ∈ pluralFrame.staticWriteBindings}
end proc;
```

10.5.5 Environmental Lookup

findThis(*env*, *allowPrototypeThis*) returns the value of `this`. If *allowPrototypeThis* is **true**, allow `this` to be defined by either an instance member of a class or a `prototype` function. If *allowPrototypeThis* is **false**, allow `this` to be defined only by an instance member of a class.

```
proc findThis(env: ENVIRONMENT, allowPrototypeThis: BOOLEAN): OBJECTIOPT
    for each frame ∈ env do
        if frame ∈ PARAMETERFRAME and frame.this ≠ none then
            if allowPrototypeThis or not frame.prototype then return frame.this end if
        end if
    end for each;
    return none
end proc;

proc lexicalRead(env: ENVIRONMENT, multiname: MULTINAME, phase: PHASE): OBJECT
    kind: LOOKUPKIND ← LEXICALLOOKUP[this: findThis(env, false)];
    i: INTEGER ← 0;
    while i < |env| do
        frame: FRAME ← env[i];
        result: OBJECTOPT ← readProperty(frame, multiname, kind, phase);
        if result ≠ none then return result end if;
        i ← i + 1
    end while;
    throw referenceError
end proc;

proc lexicalWrite(env: ENVIRONMENT, multiname: MULTINAME, newValue: OBJECT, createIfMissing: BOOLEAN,
        phase: {run})
    kind: LOOKUPKIND ← LEXICALLOOKUP[this: findThis(env, false)];
    i: INTEGER ← 0;
    while i < |env| do
        frame: FRAME ← env[i];
        result: {none, ok} ← writeProperty(frame, multiname, kind, false, newValue, phase);
        if result = ok then return end if;
        i ← i + 1
    end while;
    if createIfMissing then
        g: PACKAGE ∪ GLOBAL ← getPackageOrGlobalFrame(env);
        if g ∈ GLOBAL then
            Now try to write the variable into g again, this time allowing new dynamic bindings to be created dynamically.
            result: {none, ok} ← writeProperty(g, multiname, kind, true, newValue, phase);
            if result = ok then return end if
        end if
    end if;
    throw referenceError
end proc;

proc lexicalDelete(env: ENVIRONMENT, multiname: MULTINAME, phase: {run}): BOOLEAN
    kind: LOOKUPKIND ← LEXICALLOOKUP[this: findThis(env, false)];
    i: INTEGER ← 0;
    while i < |env| do
        frame: FRAME ← env[i];
        result: BOOLEANOPT ← deleteProperty(frame, multiname, kind, phase);
        if result ≠ none then return result end if;
        i ← i + 1
    end while;
    return true
end proc;
```

10.5.6 Property Lookup

```
tag propertyLookup;
```

tuple LEXICALLOOKUP
  this: OBJECTIOPT
end tuple;

LOOKUPKIND = {**propertyLookup**} ∪ LEXICALLOOKUP;

proc *selectPublicName*(*multiname*: MULTINAME): STRINGOPT
  if some *qname* ∈ *multiname* **satisfies** *qname*.namespace = *publicNamespace* **then**
    return *qname*.id
  end if;
  return **none**
end proc;

proc *findFlatMember*(*frame*: FRAME, *multiname*: MULTINAME, *access*: {**read**, **write**}, *phase*: PHASE):
    STATICMEMBEROPT
  matchingBindings: STATICBINDING{} ←
      {*b* | ∃*b* ∈ *staticBindingsWithAccess*(*frame*, *access*) **such that** *b*.qname ∈ *multiname*};
  if *matchingBindings* = {} **then return** **none** **end if**;
  matchingMembers: STATICMEMBER{} ← {*b*.content | ∃*b* ∈ *matchingBindings*};
  Note that if the same member was found via several different bindings *b*, then it will appear only once in the set *matchingMembers*.
  if |*matchingMembers*| > 1 **then**
    This access is ambiguous because the bindings it found belong to several different members in the same class.
    throw **propertyAccessError**
  end if;
  return the one element of *matchingMembers*
end proc;

proc *findStaticMember*(*c*: CLASSOPT, *multiname*: MULTINAME, *access*: {**read**, **write**}, *phase*: PHASE):
{**none**} ∪ STATICMEMBER ∪ QUALIFIEDNAME
s: CLASSOPT ← *c*;
while *s* ≠ **none** **do**
matchingStaticBindings: STATICBINDING{} ←
{*b* | ∃*b* ∈ *staticBindingsWithAccess*(*s*, *access*) **such that** *b*.qname ∈ *multiname*};
Note that if the same member was found via several different bindings *b*, then it will appear only once in the set *matchingStaticMembers*.
matchingStaticMembers: STATICMEMBER{} ← {*b*.content | ∃*b* ∈ *matchingStaticBindings*};
if *matchingStaticMembers* ≠ {} **then**
if |*matchingStaticMembers*| = 1 **then**
return the one element of *matchingStaticMembers*
else
This access is ambiguous because the bindings it found belong to several different static members in the same class.
throw **propertyAccessError**
end if
end if;
If a static member wasn't found in a class, look for an instance member in that class as well.
matchingInstanceBindings: INSTANCEBINDING{} ← {*b* | ∃*b* ∈ *instanceBindingsWithAccess*(*s*, *access*) **such that** *b*.qname ∈ *multiname*};
Note that if the same INSTANCEMEMBER was found via several different bindings *b*, then it will appear only once in the set *matchingInstanceMembers*.
matchingInstanceMembers: INSTANCEMEMBER{} ← {*b*.content | ∃*b* ∈ *matchingInstanceBindings*};
if *matchingInstanceMembers* ≠ {} **then**
if |*matchingInstanceMembers*| = 1 **then**
Return the qualified name of any matching binding. It doesn't matter which because they all refer to the same INSTANCEMEMBER, and if one is overridden by a subclass then all must be overridden in the same way by that subclass.
b: INSTANCEBINDING ← any element of *matchingInstanceBindings*;
return *b*.qname
else
This access is ambiguous because the bindings it found belong to several different members in the same class.
throw **propertyAccessError**
end if
end if;
s ← *s*.super
end while;
return **none**
end proc;

proc *resolveInstanceMemberName*(*c*: CLASS, *multiname*: MULTINAME, *access*: {**read**, **write**}, *phase*: PHASE): QUALIFIEDNAMEOPT

Start from the root class (`Object`) and proceed through more specific classes that are ancestors of *c*.

for each *s* ∈ *ancestors*(*c*) **do**

matchingInstanceBindings: INSTANCEBINDING{} ← {*b* | ∃*b* ∈ *instanceBindingsWithAccess*(*s*, *access*) **such that** *b*.qname ∈ *multiname*};

Note that if the same INSTANCEMEMBER was found via several different bindings *b*, then it will appear only once in the set *matchingMembers*.

matchingInstanceMembers: INSTANCEMEMBER{} ← {*b*.content | ∃*b* ∈ *matchingInstanceBindings*};

if *matchingInstanceMembers* ≠ {} **then**

if |*matchingInstanceMembers*| = 1 **then**

Return the qualified name of any matching binding. It doesn't matter which because they all refer to the same INSTANCEMEMBER, and if one is overridden by a subclass then all must be overridden in the same way by that subclass.

b: INSTANCEBINDING ← any element of *matchingInstanceBindings*;

return *b*.qname

else

This access is ambiguous because the bindings it found belong to several different members in the same class.

throw propertyAccessError

end if

end if

end for each;

return none

end proc;

proc *findInstanceMember*(*c*: CLASSOPT, *qname*: QUALIFIEDNAMEOPT, *access*: {**read**, **write**}): INSTANCEMEMBEROPT

if *qname* = **none then return none end if**;

s: CLASSOPT ← *c*;

while *s* ≠ **none do**

if some *b* ∈ *instanceBindingsWithAccess*(*s*, *access*) **satisfies** *b*.qname = *qname* **then**

return *b*.content

end if;

s ← *s*.super

end while;

return none

end proc;

10.5.7 Reading a Property

tag generic;

```
proc readProperty(container: ObjOptionalLimit ∪ Frame, multiname: Multiname, kind: LookupKind,
      phase: Phase): ObjectOpt
   case container of
      Undefined ∪ Null ∪ Boolean ∪ Float64 ∪ Long ∪ ULong ∪ Character ∪ String ∪ Namespace ∪
            CompoundAttribute ∪ MethodClosure ∪ Instance do
         c: Class ← objectType(container);
         qname: QualifiedNameOpt ← resolveInstanceMemberName(c, multiname, read, phase);
         if qname = none and container ∈ DynamicInstance then
            return readDynamicProperty(container, multiname, kind, phase)
         else return readInstanceMember(container, c, qname, phase)
         end if;
      SystemFrame ∪ Global ∪ Package ∪ ParameterFrame ∪ BlockFrame do
         m: StaticMemberOpt ← findFlatMember(container, multiname, read, phase);
         if m = none and container ∈ Global then
            return readDynamicProperty(container, multiname, kind, phase)
         else return readStaticMember(m, phase)
         end if;
      Class do
         this: Object ← {inaccessible, none, generic};
         case kind of
            {propertyLookup} do this ← generic;
            LexicalLookup do this ← kind.this
         end case;
         m2: {none} ∪ StaticMember ∪ QualifiedName ← findStaticMember(container, multiname, read, phase);
         if m2 ∈ QualifiedName then return readStaticMember(m2, phase) end if;
         case this of
            {none} do throw propertyAccessError;
            {inaccessible} do throw compileExpressionError;
            {generic} do ????;
            Object do return readInstanceMember(this, objectType(this), m2, phase)
         end case;
      Prototype do return readDynamicProperty(container, multiname, kind, phase);
      LimitedInstance do
         superclass: ClassOpt ← container.limit.super;
         if superclass = none then return none end if;
         qname: QualifiedNameOpt ← resolveInstanceMemberName(superclass, multiname, read, phase);
         return readInstanceMember(container.instance, superclass, qname, phase)
   end case
end proc;

proc readInstanceMember(this: Object, c: Class, qname: QualifiedNameOpt, phase: Phase): ObjectOpt
   m: InstanceMemberOpt ← findInstanceMember(c, qname, read);
   case m of
      {none} do return none;
      InstanceVariable do
         if phase = compile and not m.immutable then throw compileExpressionError
         end if;
         v: ObjectU ← findSlot(this, m).value;
         if v = uninitialised then throw uninitialisedError end if;
         return v;
      InstanceMethod do return MethodClosure[this: this, method: m];
      InstanceGetter do return m.call(this, m.env, phase);
      InstanceSetter do
         m cannot be an InstanceSetter because these are only represented as write-only members.
   end case
end proc;
```

```
proc readStaticMember(m: STATICMEMBEROPT, phase: PHASE): OBJECTOPT
    case m of
        {none} do return none;
        {forbidden} do throw propertyAccessError;
        VARIABLE do return readVariable(m, phase);
        HOISTEDVAR do
            if phase = compile then throw compileExpressionError end if;
            value: OBJECT ∪ OPENINSTANCE ← m.value;
            Note that value can be an OPENINSTANCE only during the compile phase, which was ruled out above.
            return value;
        CONSTRUCTORMETHOD do return m.code;
        GETTER do
            env: ENVIRONMENTI ← m.env;
            if env = inaccessible then throw compileExpressionError end if;
            return m.call(env, phase);
        SETTER do
            m cannot be a SETTER because these are only represented as write-only members.
    end case
end proc;

proc readDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME, kind: LOOKUPKIND, phase: PHASE):
        OBJECTOPT
    name: STRINGOPT ← selectPublicName(multiname);
    if name = none then return none end if;
    if phase = compile then throw compileExpressionError end if;
    if some dp ∈ container.dynamicProperties satisfies dp.name = name then
        return dp.value
    end if;
    if container ∈ PROTOTYPE then
        parent: PROTOTYPEOPT ← container.parent;
        if parent ≠ none then return readDynamicProperty(parent, multiname, kind, phase)
        end if
    end if;
    if kind = propertyLookup then return undefined end if;
    return none
end proc;
```

```
proc readVariable(v: VARIABLE, phase: PHASE): OBJECT
    if phase = compile and not v.immutable then throw compileExpressionError end if;
    value: VARIABLEVALUE ← v.value;
    case value of
        OBJECT do return value;
        {inaccessible} do
            if phase = compile then throw compileExpressionError
            else throw uninitialisedError
            end if;
        {uninitialised} do throw uninitialisedError;
        OPENINSTANCE do
            Note that an uninstantiated function can only be found when phase = compile.
            throw compileExpressionError;
        () → OBJECT do
            Note that phase = compile because all futures are resolved by the end of the compilation phase.
            v.value ← inaccessible;
            type: CLASS ← getVariableType(v, phase);
            newValue: OBJECT ← value();
            coercedValue: OBJECT ← assignmentConversion(newValue, type);
            v.value ← coercedValue;
            return newValue
    end case
end proc;
```

10.5.8 Writing a Property

```
proc writeProperty(container: OBJOPTIONALLIMIT ∪ FRAME, multiname: MULTINAME, kind: LOOKUPKIND,
      createIfMissing: BOOLEAN, newValue: OBJECT, phase: {run}): {none, ok}
   case container of
      UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ LONG ∪ ULONG ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪
            COMPOUNDATTRIBUTE ∪ METHODCLOSURE do
         return none;
      SYSTEMFRAME ∪ GLOBAL ∪ PACKAGE ∪ PARAMETERFRAME ∪ BLOCKFRAME do
         m: STATICMEMBEROPT ← findFlatMember(container, multiname, write, phase);
         if m = none and container ∈ GLOBAL then
            return writeDynamicProperty(container, multiname, createIfMissing, newValue, phase)
         else return writeStaticMember(m, newValue, phase)
         end if;
      CLASS do
         this: OBJECTIOPT;
         case kind of
            {propertyLookup} do this ← none;
            LEXICALLOOKUP do this ← kind.this
         end case;
         m2: {none} ∪ STATICMEMBER ∪ QUALIFIEDNAME ← findStaticMember(container, multiname, write, phase);
         if m2 ∈ QUALIFIEDNAME then return writeStaticMember(m2, newValue, phase)
         elsif this = none then throw propertyAccessError
         elsif this = inaccessible then throw compileExpressionError
         else return writeInstanceMember(this, objectType(this), m2, newValue, phase)
         end if;
      PROTOTYPE do
         return writeDynamicProperty(container, multiname, createIfMissing, newValue, phase);
      INSTANCE do
         c: CLASS ← objectType(container);
         qname: QUALIFIEDNAMEOPT ← resolveInstanceMemberName(objectType(container), multiname, write, phase);
         if qname = none and container ∈ DYNAMICINSTANCE then
            return writeDynamicProperty(container, multiname, createIfMissing, newValue, phase)
         else return writeInstanceMember(container, c, qname, newValue, phase)
         end if;
      LIMITEDINSTANCE do
         superclass: CLASSOPT ← container.limit.super;
         if superclass = none then return none end if;
         qname: QUALIFIEDNAMEOPT ← resolveInstanceMemberName(superclass, multiname, write, phase);
         return writeInstanceMember(container.instance, superclass, qname, newValue, phase)
   end case
end proc;
```

```
proc writeInstanceMember(this: OBJECT, c: CLASS, qname: QUALIFIEDNAMEOPT, newValue: OBJECT, phase: {run}):
        {none, ok}
    m: INSTANCEMEMBEROPT ← findInstanceMember(c, qname, write);
    case m of
        {none} do return none;
        INSTANCEVARIABLE do
            s: SLOT ← findSlot(this, m);
            if m.immutable and s.value ≠ uninitialised then throw propertyAccessError
            end if;
            coercedValue: OBJECT ← assignmentConversion(newValue, m.type);
            s.value ← coercedValue;
            return ok;
        INSTANCEMETHOD do throw propertyAccessError;
        INSTANCEGETTER do
            m cannot be an INSTANCEGETTER because these are only represented as read-only members.
        INSTANCESETTER do
            coercedValue: OBJECT ← assignmentConversion(newValue, m.type);
            m.call(this, coercedValue, m.env, phase);
            return ok
    end case
end proc;

proc writeStaticMember(m: STATICMEMBEROPT, newValue: OBJECT, phase: {run}): {none, ok}
    case m of
        {none} do return none;
        {forbidden} ∪ CONSTRUCTORMETHOD do throw propertyAccessError;
        VARIABLE do writeVariable(m, newValue, phase); return ok;
        HOISTEDVAR do m.value ← newValue; return ok;
        GETTER do
            m cannot be a GETTER because these are only represented as read-only members.
        SETTER do
            coercedValue: OBJECT ← assignmentConversion(newValue, m.type);
            env: ENVIRONMENTI ← m.env;
            Note that all instances are resolved for the run phase, so env ≠ inaccessible.
            m.call(coercedValue, env, phase);
            return ok
    end case
end proc;
```

```
proc writeDynamicProperty(container: DynamicObject, multiname: Multiname, createIfMissing: Boolean,
        newValue: Object, phase: {run}): {none, ok}
    name: StringOpt ← selectPublicName(multiname);
    if name = none then return none end if;
    if some dp ∈ container.dynamicProperties satisfies dp.name = name then
        dp.value ← newValue;
        return ok
    end if;
    if not createIfMissing then return none end if;
    Before trying to create a new dynamic property, check that there is no read-only fixed property with the same name.
    m: {none} ∪ StaticMember ∪ QualifiedName;
    case container of
        Prototype do m ← none;
        DynamicInstance do
            m ← resolveInstanceMemberName(objectType(container), multiname, read, phase);
        Global do m ← findFlatMember(container, multiname, read, phase)
    end case;
    if m ≠ none then return none end if;
    container.dynamicProperties ←
            container.dynamicProperties ∪ {new DynamicProperty⟦name: name, value: newValue⟧};
    return ok
end proc;

proc getVariableType(v: Variable, phase: Phase): Class
    type: VariableType ← v.type;
    case type of
        Class do return type;
        {inaccessible} do
            Note that this can only happen when phase = compile because the compilation phase ensures that all types are
                valid, so invalid types will not occur during the run phase.
            throw compileExpressionError;
        () ∈ Class do
            Note that phase = compile because all futures are resolved by the end of the compilation phase.
            v.type ← inaccessible;
            newType: Class ← type();
            v.type ← newType;
            return newType
    end case
end proc;

proc writeVariable(v: Variable, newValue: Object, phase: {run})
    type: Class ← getVariableType(v, phase);
    if v.value = inaccessible or (v.immutable and v.value ≠ uninitialised) then
        throw propertyAccessError
    end if;
    coercedValue: Object ← assignmentConversion(newValue, type);
    v.value ← coercedValue
end proc;
```

10.5.9 Deleting a Property

```
proc deleteProperty(container: OBJOPTIONALLIMIT ∪ FRAME, multiname: MULTINAME, kind: LOOKUPKIND,
      phase: {run}): BOOLEANOPT
   case container of
      UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ LONG ∪ ULONG ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪
            COMPOUNDATTRIBUTE ∪ METHODCLOSURE ∪ INSTANCE do
         c: CLASS ← objectType(container);
         qname: QUALIFIEDNAMEOPT ← resolveInstanceMemberName(c, multiname, read, phase);
         if qname = none and container ∈ DYNAMICINSTANCE then
            return deleteDynamicProperty(container, multiname)
         else return deleteInstanceMember(c, qname)
         end if;
      SYSTEMFRAME ∪ GLOBAL ∪ PACKAGE ∪ PARAMETERFRAME ∪ BLOCKFRAME do
         m: STATICMEMBEROPT ← findFlatMember(container, multiname, read, phase);
         if m = none and container ∈ GLOBAL then
            return deleteDynamicProperty(container, multiname)
         else return deleteStaticMember(m)
         end if;
      CLASS do
         this: OBJECT ∪ {none, generic};
         case kind of
            {propertyLookup} do this ← generic;
            LEXICALLOOKUP do
               this ← kind.this;
               Note that this cannot be inaccessible during the run phase.
         end case;
         m2: {none} ∪ STATICMEMBER ∪ QUALIFIEDNAME ← findStaticMember(container, multiname, read, phase);
         if m2 ∈ QUALIFIEDNAME then return deleteStaticMember(m2) end if;
         case this of
            {none} do throw propertyAccessError;
            {generic} do return false;
            OBJECT do return deleteInstanceMember(objectType(this), m2)
         end case;
      PROTOTYPE do return deleteDynamicProperty(container, multiname);
      LIMITEDINSTANCE do
         superclass: CLASSOPT ← container.limit.super;
         if superclass = none then return none end if;
         qname: QUALIFIEDNAMEOPT ← resolveInstanceMemberName(superclass, multiname, read, phase);
         return deleteInstanceMember(superclass, qname)
   end case
end proc;

proc deleteInstanceMember(c: CLASS, qname: QUALIFIEDNAMEOPT): BOOLEANOPT
   m: INSTANCEMEMBEROPT ← findInstanceMember(c, qname, read);
   if m = none then return none end if;
   return false
end proc;

proc deleteStaticMember(m: STATICMEMBEROPT): BOOLEANOPT
   case m of
      {none} do return none;
      {forbidden} do throw propertyAccessError;
      VARIABLE ∪ HOISTEDVAR ∪ CONSTRUCTORMETHOD ∪ GETTER ∪ SETTER do return false
   end case
end proc;
```

```
proc deleteDynamicProperty(container: DynamicObject, multiname: Multiname): BooleanOpt
    name: StringOpt ← selectPublicName(multiname);
    if name = none then return none end if;
    if some dp ∈ container.dynamicProperties satisfies dp.name = name then
        container.dynamicProperties ← container.dynamicProperties – {dp};
        return true
    else return none
    end if
end proc;
```

10.6 Invocation

```
proc badConstruct(args: ArgumentList, runtimeEnv: Environment, phase: Phase): Object
    throw propertyAccessError
end proc;
```

10.7 Pre-Evaluation

preEvaluators: (() → ())[] ← [];

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*:

β ∈ {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 9.3).

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] = 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 and Evaluation

```
proc Validate[Qualifier] (cxt: CONTEXT, env: ENVIRONMENT): NAMESPACE
    [Qualifier ⊆ Identifier] do
        multiname: MULTINAME ⊆ {QUALIFIEDNAME⊆namespace: ns, id: Name[Identifier]⊆|
                ⊆ns ⊆ cxt.openNamespaces};
        a: OBJECT ⊆ lexicalRead(env, multiname, compile);
        if a ⊆ NAMESPACE then throw badValueError end if;
        return a;
    [Qualifier ⊆ public] do return publicNamespace;
    [Qualifier ⊆ private] do
        c: CLASSOPT ⊆ getEnclosingClass(env);
        if c = none then throw syntaxError end if;
        return c.privateNamespace
end proc;

Multiname[SimpleQualifiedIdentifier]: MULTINAME;
```

```
proc Validate[SimpleQualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
    [SimpleQualifiedIdentifier ⇒ Identifier] do
        multiname: MULTINAME ← {QUALIFIEDNAME[namespace: ns, id: Name[Identifier]] |
                ∈ ns ∈ cxt.openNamespaces};
        Multiname[SimpleQualifiedIdentifier] ← multiname;
    [SimpleQualifiedIdentifier ⇒ Qualifier :: Identifier] do
        q: NAMESPACE ← Validate[Qualifier](cxt, env);
        Multiname[SimpleQualifiedIdentifier] ← {QUALIFIEDNAME[namespace: q, id: Name[Identifier]]}
end proc;

Multiname[ExpressionQualifiedIdentifier]: MULTINAME;

proc Validate[ExpressionQualifiedIdentifier ⇒ ParenExpression :: Identifier] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[ParenExpression](cxt, env);
    r: OBJORREF ← Eval[ParenExpression](env, compile);
    q: OBJECT ← readReference(r, compile);
    if q ∉ NAMESPACE then throw badValueError end if;
    Multiname[ExpressionQualifiedIdentifier] ← {QUALIFIEDNAME[namespace: q, id: Name[Identifier]]}
end proc;

Multiname[QualifiedIdentifier]: MULTINAME;

proc Validate[QualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
    [QualifiedIdentifier ⇒ SimpleQualifiedIdentifier] do
        Validate[SimpleQualifiedIdentifier](cxt, env);
        Multiname[QualifiedIdentifier] ← Multiname[SimpleQualifiedIdentifier];
    [QualifiedIdentifier ⇒ ExpressionQualifiedIdentifier] do
        Validate[ExpressionQualifiedIdentifier](cxt, env);
        Multiname[QualifiedIdentifier] ← Multiname[ExpressionQualifiedIdentifier]
end proc;
```

12.3 Primary Expressions

Syntax

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

ParenExpression ⇒ ( AssignmentExpression^allowIn )

ParenListExpression ⇒
    ParenExpression
  | ( ListExpression^allowIn , AssignmentExpression^allowIn )
```

Validation

proc Validate[*PrimaryExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*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**
if *findThis*(*env*, **true**) = **none then throw syntaxError end if**;
[*PrimaryExpression* ⇒ **RegularExpression**] **do nothing**;
[*PrimaryExpression* ⇒ *ParenListExpression*] **do**
Validate[*ParenListExpression*](*cxt*, *env*);
[*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
[*PrimaryExpression* ⇒ *ObjectLiteral*] **do** Validate[*ObjectLiteral*](*cxt*, *env*);
[*PrimaryExpression* ⇒ *FunctionExpression*] **do** Validate[*FunctionExpression*](*cxt*, *env*)
end proc;

Validate[*ParenExpression* ⇒ **(** *AssignmentExpression*^{allowIn} **)**]: CONTEXT × ENVIRONMENT → ()
= Validate[*AssignmentExpression*^{allowIn}];

proc Validate[*ParenListExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ParenListExpression* ⇒ *ParenExpression*] **do** Validate[*ParenExpression*](*cxt*, *env*);
[*ParenListExpression* ⇒ **(** *ListExpression*^{allowIn} **,** *AssignmentExpression*^{allowIn} **)**] **do**
Validate[*ListExpression*^{allowIn}](*cxt*, *env*);
Validate[*AssignmentExpression*^{allowIn}](*cxt*, *env*)
end proc;

Evaluation

proc Eval[*PrimaryExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): 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** Value[**Number**];
[*PrimaryExpression* ⇒ **String**] **do return** Value[**String**];
[*PrimaryExpression* ⇒ **this**] **do**
this: OBJECTIOPT ← *findThis*(*env*, **true**);
Note that Validate ensured that *this* cannot be **none** at this point.
if *this* = **inaccessible then throw compileExpressionError end if**;
return *this*;
[*PrimaryExpression* ⇒ **RegularExpression**] **do** ????;
[*PrimaryExpression* ⇒ *ParenListExpression*] **do**
return Eval[*ParenListExpression*](*env*, *phase*);
[*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
[*PrimaryExpression* ⇒ *ObjectLiteral*] **do return** Eval[*ObjectLiteral*](*env*, *phase*);
[*PrimaryExpression* ⇒ *FunctionExpression*] **do**
return Eval[*FunctionExpression*](*env*, *phase*)
end proc;

Eval[*ParenExpression* ⊡ (*AssignmentExpression*^{allowIn})]: ENVIRONMENT ⊡ PHASE ⊡ OBJORREF
= Eval[*AssignmentExpression*^{allowIn}];

proc Eval[*ParenListExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*ParenListExpression* ⊡ *ParenExpression*] **do return** Eval[*ParenExpression*](*env*, *phase*);
[*ParenListExpression* ⊡ (*ListExpression*^{allowIn} , *AssignmentExpression*^{allowIn})] **do**
ra: OBJORREF ⊡ Eval[*ListExpression*^{allowIn}](*env*, *phase*);
readReference(*ra*, *phase*);
rb: OBJORREF ⊡ Eval[*AssignmentExpression*^{allowIn}](*env*, *phase*);
return *readReference*(*rb*, *phase*)
end proc;

proc EvalAsList[*ParenListExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJECT[]
[*ParenListExpression* ⊡ *ParenExpression*] **do**
r: OBJORREF ⊡ Eval[*ParenExpression*](*env*, *phase*);
elt: OBJECT ⊡ *readReference*(*r*, *phase*);
return [*elt*];
[*ParenListExpression* ⊡ (*ListExpression*^{allowIn} , *AssignmentExpression*^{allowIn})] **do**
elts: OBJECT[] ⊡ EvalAsList[*ListExpression*^{allowIn}](*env*, *phase*);
r: OBJORREF ⊡ Eval[*AssignmentExpression*^{allowIn}](*env*, *phase*);
elt: OBJECT ⊡ *readReference*(*r*, *phase*);
return *elts* ⊕ [*elt*]
end proc;

12.4 Function Expressions

Syntax

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

Validation

proc Validate[*FunctionExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*FunctionExpression* ⊡ `function` *FunctionSignature Block*] **do** ????;
[*FunctionExpression* ⊡ `function` *Identifier FunctionSignature Block*] **do** ????
end proc;

Evaluation

proc Eval[*FunctionExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*FunctionExpression* ⊡ `function` *FunctionSignature Block*] **do** ????;
[*FunctionExpression* ⊡ `function` *Identifier FunctionSignature Block*] **do** ????
end proc;

12.5 Object Literals

Syntax

ObjectLiteral ⊡
`{ }`
| `{` *FieldList* `}`

FieldList ⊡
  LiteralField
 | *FieldList* , *LiteralField*

LiteralField ⊡ *FieldName* : *AssignmentExpression*[allowIn]

FieldName ⊡
  Identifier
 | **String**
 | **Number**

Validation

proc Validate[*ObjectLiteral*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*ObjectLiteral* ⊡ { }] **do nothing**;
  [*ObjectLiteral* ⊡ { *FieldList* }] **do** Validate[*FieldList*](*cxt*, *env*)
end proc;

proc Validate[*FieldList*] (*cxt*: CONTEXT, *env*: ENVIRONMENT): STRING{}
  [*FieldList* ⊡ *LiteralField*] **do return** Validate[*LiteralField*](*cxt*, *env*);
  [*FieldList*$_0$ ⊡ *FieldList*$_1$, *LiteralField*] **do**
    names1: STRING{} ⊡ Validate[*FieldList*$_1$](*cxt*, *env*);
    names2: STRING{} ⊡ Validate[*LiteralField*](*cxt*, *env*);
    if *names1* ⊡ *names2* ≠ {} **then throw syntaxError end if**;
    return *names1* ⊡ *names2*
end proc;

proc Validate[*LiteralField* ⊡ *FieldName* : *AssignmentExpression*[allowIn]] (*cxt*: CONTEXT, *env*: ENVIRONMENT): STRING{}
  names: STRING{} ⊡ Validate[*FieldName*](*cxt*, *env*);
  Validate[*AssignmentExpression*[allowIn]](*cxt*, *env*);
  return *names*
end proc;

proc Validate[*FieldName*] (*cxt*: CONTEXT, *env*: ENVIRONMENT): STRING{}
  [*FieldName* ⊡ *Identifier*] **do return** {Name[*Identifier*]};
  [*FieldName* ⊡ **String**] **do return** {Value[**String**]};
  [*FieldName* ⊡ **Number**] **do return** {*float64ToString*(Value[**Number**])}
end proc;

Evaluation

proc Eval[*ObjectLiteral*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
  [*ObjectLiteral* ⊡ { }] **do**
    if *phase* = **compile then throw compileExpressionError end if**;
    return new PROTOTYPE⊡parent: *objectPrototype*, dynamicProperties: {}⊡
  [*ObjectLiteral* ⊡ { *FieldList* }] **do**
    if *phase* = **compile then throw compileExpressionError end if**;
    properties: DYNAMICPROPERTY{} ⊡ Eval[*FieldList*](*env*, *phase*);
    return new PROTOTYPE⊡parent: *objectPrototype*, dynamicProperties: *properties*⊡
end proc;

proc Eval[*FieldList*] (*env*: ENVIRONMENT, *phase*: PHASE): DYNAMICPROPERTY{}
  [*FieldList* ⊡ *LiteralField*] **do**
    na: NAMEDARGUMENT ⊡ Eval[*LiteralField*](*env*, *phase*);
    return {**new** DYNAMICPROPERTY⊡name: *na*.name, value: *na*.value⊡};

```
    [FieldList0 ⊆ FieldList1 , LiteralField] do
        properties: DynamicProperty{} ⊆ Eval[FieldList1](env, phase);
        na: NamedArgument ⊆ Eval[LiteralField](env, phase);
        if some p ∈ properties satisfies p.name = na.name then
            throw argumentMismatchError
        end if;
        return properties ∪ {new DynamicProperty[name: na.name, value: na.value]}
end proc;

proc Eval[LiteralField ⊆ FieldName : AssignmentExpression^allowIn]
        (env: Environment, phase: Phase): NamedArgument
    name: String ⊆ Eval[FieldName](env, phase);
    r: ObjOrRef ⊆ Eval[AssignmentExpression^allowIn](env, phase);
    value: Object ⊆ readReference(r, phase);
    return NamedArgument[name: name, value: value]
end proc;

proc Eval[FieldName] (env: Environment, phase: Phase): String
    [FieldName ⊆ Identifier] do return Name[Identifier];
    [FieldName ⊆ String] do return Value[String];
    [FieldName ⊆ Number] do return float64ToString(Value[Number])
end proc;
```

12.6 Array Literals

Syntax

```
ArrayLiteral ⊆ [ ElementList ]

ElementList ⊆
      LiteralElement
    | ElementList , LiteralElement

LiteralElement ⊆
      «empty»
    | AssignmentExpression^allowIn
```

12.7 Super Expressions

Syntax

```
SuperExpression ⊆
      super
    | super ParenExpression
```

Validation

```
proc Validate[SuperExpression] (cxt: Context, env: Environment)
    [SuperExpression ⊆ super] do
        if getEnclosingClass(env) = none or findThis(env, false) = none then
            throw syntaxError
        end if;
```

```
    [SuperExpression ≡ super ParenExpression] do
        if getEnclosingClass(env) = none then throw syntaxError end if;
        Validate[ParenExpression](cxt, env)
end proc;
```

Evaluation

```
proc Eval[SuperExpression] (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT
    [SuperExpression ≡ super] do
        this: OBJECTIOPT ← findThis(env, false);
        Note that Validate ensured that this cannot be none at this point.
        if this = inaccessible then throw compileExpressionError end if;
        limit: CLASSOPT ← getEnclosingClass(env);
        Note that Validate ensured that limit cannot be none at this point.
        return readLimitedReference(this, limit, phase);
    [SuperExpression ≡ super ParenExpression] do
        r: OBJORREF ← Eval[ParenExpression](env, phase);
        limit: CLASSOPT ← getEnclosingClass(env);
        Note that Validate ensured that limit cannot be none at this point.
        return readLimitedReference(r, limit, phase)
end proc;
```

readLimitedReference(*r*, *phase*) reads the reference, if any, inside *r* and returns the result, retaining *limit*. The object read from the reference is checked to make sure that it is an instance of *limit* or one of its descendants. If *phase* is **compile**, only compile-time expressions can be evaluated in the process of reading *r*.

```
proc readLimitedReference(r: OBJORREF, limit: CLASS, phase: PHASE): OBJOPTIONALLIMIT
    o: OBJECT ← readReference(r, phase);
    if o = null then return null end if;
    if o ∉ INSTANCE or not hasType(o, limit) then throw badValueError end if;
    return LIMITEDINSTANCE{instance: o, limit: limit}
end proc;
```

12.8 Postfix Expressions

Syntax

```
PostfixExpression ≡
    AttributeExpression
  | FullPostfixExpression
  | ShortNewExpression

AttributeExpression ≡
    SimpleQualifiedIdentifier
  | AttributeExpression MemberOperator
  | AttributeExpression Arguments

FullPostfixExpression ≡
    PrimaryExpression
  | ExpressionQualifiedIdentifier
  | FullNewExpression
  | FullPostfixExpression MemberOperator
  | SuperExpression DotOperator
  | FullPostfixExpression Arguments
  | PostfixExpression [no line break] ++
  | PostfixExpression [no line break] --
```

FullNewExpression ⇒ **new** *FullNewSubexpression Arguments*

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

ShortNewExpression ⇒ **new** *ShortNewSubexpression*

ShortNewSubexpression ⇒
  FullNewSubexpression
| *ShortNewExpression*

Validation

Validate[*PostfixExpression*]: CONTEXT × ENVIRONMENT → ();
  Validate[*PostfixExpression* ⇒ *AttributeExpression*] = Validate[*AttributeExpression*];
  Validate[*PostfixExpression* ⇒ *FullPostfixExpression*] = Validate[*FullPostfixExpression*];
  Validate[*PostfixExpression* ⇒ *ShortNewExpression*] = Validate[*ShortNewExpression*];

Strict[*AttributeExpression*]: BOOLEAN;

proc Validate[*AttributeExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*AttributeExpression* ⇒ *SimpleQualifiedIdentifier*] **do**
    Validate[*SimpleQualifiedIdentifier*](*cxt*, *env*);
    Strict[*AttributeExpression*] ← *cxt*.strict;
  [*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *MemberOperator*] **do**
    Validate[*AttributeExpression*$_1$](*cxt*, *env*);
    Validate[*MemberOperator*](*cxt*, *env*);
  [*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *Arguments*] **do**
    Validate[*AttributeExpression*$_1$](*cxt*, *env*);
    Validate[*Arguments*](*cxt*, *env*)
end proc;

Strict[*FullPostfixExpression*]: BOOLEAN;

proc Validate[*FullPostfixExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*FullPostfixExpression* ⇒ *PrimaryExpression*] **do**
    Validate[*PrimaryExpression*](*cxt*, *env*);
  [*FullPostfixExpression* ⇒ *ExpressionQualifiedIdentifier*] **do**
    Validate[*ExpressionQualifiedIdentifier*](*cxt*, *env*);
    Strict[*FullPostfixExpression*] ← *cxt*.strict;
  [*FullPostfixExpression* ⇒ *FullNewExpression*] **do**
    Validate[*FullNewExpression*](*cxt*, *env*);
  [*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *MemberOperator*] **do**
    Validate[*FullPostfixExpression*$_1$](*cxt*, *env*);
    Validate[*MemberOperator*](*cxt*, *env*);
  [*FullPostfixExpression* ⇒ *SuperExpression DotOperator*] **do**
    Validate[*SuperExpression*](*cxt*, *env*);
    Validate[*DotOperator*](*cxt*, *env*);
  [*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *Arguments*] **do**
    Validate[*FullPostfixExpression*$_1$](*cxt*, *env*);
    Validate[*Arguments*](*cxt*, *env*);

```
    [FullPostfixExpression ⇒ PostfixExpression [no line break] ++] do
        Validate[PostfixExpression](cxt, env);
    [FullPostfixExpression ⇒ PostfixExpression [no line break] --] do
        Validate[PostfixExpression](cxt, env)
end proc;

proc Validate[FullNewExpression ⇒ new FullNewSubexpression Arguments] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[FullNewSubexpression](cxt, env);
    Validate[Arguments](cxt, env)
end proc;

Strict[FullNewSubexpression]: BOOLEAN;

proc Validate[FullNewSubexpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [FullNewSubexpression ⇒ PrimaryExpression] do Validate[PrimaryExpression](cxt, env);
    [FullNewSubexpression ⇒ QualifiedIdentifier] do
        Validate[QualifiedIdentifier](cxt, env);
        Strict[FullNewSubexpression] ← cxt.strict;
    [FullNewSubexpression ⇒ FullNewExpression] do Validate[FullNewExpression](cxt, env);
    [FullNewSubexpression0 ⇒ FullNewSubexpression1 MemberOperator] do
        Validate[FullNewSubexpression1](cxt, env);
        Validate[MemberOperator](cxt, env);
    [FullNewSubexpression ⇒ SuperExpression DotOperator] do
        Validate[SuperExpression](cxt, env);
        Validate[DotOperator](cxt, env)
end proc;

proc Validate[ShortNewExpression ⇒ new ShortNewSubexpression] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[ShortNewSubexpression](cxt, env)
end proc;

Validate[ShortNewSubexpression]: CONTEXT × ENVIRONMENT → ();
    Validate[ShortNewSubexpression ⇒ FullNewSubexpression] = Validate[FullNewSubexpression];
    Validate[ShortNewSubexpression ⇒ ShortNewExpression] = Validate[ShortNewExpression];
```

Evaluation

```
Eval[PostfixExpression]: ENVIRONMENT × PHASE → OBJORREF;
    Eval[PostfixExpression ⇒ AttributeExpression] = Eval[AttributeExpression];
    Eval[PostfixExpression ⇒ FullPostfixExpression] = Eval[FullPostfixExpression];
    Eval[PostfixExpression ⇒ ShortNewExpression] = Eval[ShortNewExpression];

proc Eval[AttributeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [AttributeExpression ⇒ SimpleQualifiedIdentifier] do
        return LEXICALREFERENCE{env: env, variableMultiname: Multiname[SimpleQualifiedIdentifier],
                strict: Strict[AttributeExpression]};
    [AttributeExpression0 ⇒ AttributeExpression1 MemberOperator] do
        r: OBJORREF ← Eval[AttributeExpression1](env, phase);
        a: OBJECT ← readReference(r, phase);
        return Eval[MemberOperator](env, a, phase);
```

```
    [AttributeExpression0 ⇒ AttributeExpression1 Arguments] do
        r: OBJORREF ← Eval[AttributeExpression1](env, phase);
        f: OBJECT ← readReference(r, phase);
        base: OBJECT ← referenceBase(r);
        args: ARGUMENTLIST ← Eval[Arguments](env, phase);
        return call(base, f, args, phase)
end proc;

proc Eval[FullPostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [FullPostfixExpression ⇒ PrimaryExpression] do
        return Eval[PrimaryExpression](env, phase);
    [FullPostfixExpression ⇒ ExpressionQualifiedIdentifier] do
        return LEXICALREFERENCE[env: env, variableMultiname: Multiname[ExpressionQualifiedIdentifier],
                strict: Strict[FullPostfixExpression]];
    [FullPostfixExpression ⇒ FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    [FullPostfixExpression0 ⇒ FullPostfixExpression1 MemberOperator] do
        r: OBJORREF ← Eval[FullPostfixExpression1](env, phase);
        a: OBJECT ← readReference(r, phase);
        return Eval[MemberOperator](env, a, phase);
    [FullPostfixExpression ⇒ SuperExpression DotOperator] do
        a: OBJOPTIONALLIMIT ← Eval[SuperExpression](env, phase);
        return Eval[DotOperator](env, a, phase);
    [FullPostfixExpression0 ⇒ FullPostfixExpression1 Arguments] do
        r: OBJORREF ← Eval[FullPostfixExpression1](env, phase);
        f: OBJECT ← readReference(r, phase);
        base: OBJECT ← referenceBase(r);
        args: ARGUMENTLIST ← Eval[Arguments](env, phase);
        return call(base, f, args, phase);
    [FullPostfixExpression ⇒ PostfixExpression [no line break] ++] do
        if phase = compile then throw compileExpressionError end if;
        r: OBJORREF ← Eval[PostfixExpression](env, phase);
        a: OBJECT ← readReference(r, phase);
        b: OBJECT ← plus(a, phase);
        c: OBJECT ← add(b, 1.0, phase);
        writeReference(r, c, phase);
        return b;
    [FullPostfixExpression ⇒ PostfixExpression [no line break] --] do
        if phase = compile then throw compileExpressionError end if;
        r: OBJORREF ← Eval[PostfixExpression](env, phase);
        a: OBJECT ← readReference(r, phase);
        b: OBJECT ← plus(a, phase);
        c: OBJECT ← subtract(b, 1.0, phase);
        writeReference(r, c, phase);
        return b
end proc;

proc Eval[FullNewExpression ⇒ new FullNewSubexpression Arguments]
        (env: ENVIRONMENT, phase: PHASE): OBJORREF
    r: OBJORREF ← Eval[FullNewSubexpression](env, phase);
    f: OBJECT ← readReference(r, phase);
    args: ARGUMENTLIST ← Eval[Arguments](env, phase);
    return construct(f, args, phase)
end proc;
```

```
proc Eval[FullNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [FullNewSubexpression ⊂ PrimaryExpression] do
        return Eval[PrimaryExpression](env, phase);
    [FullNewSubexpression ⊂ QualifiedIdentifier] do
        return LEXICALREFERENCE{env: env, variableMultiname: Multiname[QualifiedIdentifier],
                strict: Strict[FullNewSubexpression]};
    [FullNewSubexpression ⊂ FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    [FullNewSubexpression0 ⊂ FullNewSubexpression1 MemberOperator] do
        r: OBJORREF ← Eval[FullNewSubexpression1](env, phase);
        a: OBJECT ← readReference(r, phase);
        return Eval[MemberOperator](env, a, phase);
    [FullNewSubexpression ⊂ SuperExpression DotOperator] do
        a: OBJOPTIONALLIMIT ← Eval[SuperExpression](env, phase);
        return Eval[DotOperator](env, a, phase)
end proc;

proc Eval[ShortNewExpression ⊂ new ShortNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    r: OBJORREF ← Eval[ShortNewSubexpression](env, phase);
    f: OBJECT ← readReference(r, phase);
    return construct(f, ARGUMENTLIST{positional: [], named: {}}, phase)
end proc;

Eval[ShortNewSubexpression]: ENVIRONMENT × PHASE → OBJORREF;
    Eval[ShortNewSubexpression ⊂ FullNewSubexpression] = Eval[FullNewSubexpression];
    Eval[ShortNewSubexpression ⊂ ShortNewExpression] = Eval[ShortNewExpression];
```

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

```
proc referenceBase(r: OBJORREF): OBJECT
    case r of
        OBJECT ∪ LEXICALREFERENCE do return null;
        DOTREFERENCE ∪ BRACKETREFERENCE do
            o: OBJOPTIONALLIMIT ← r.base;
            case o of
                OBJECT do return o;
                LIMITEDINSTANCE do return o.instance
            end case
    end case
end proc;

proc call(this: OBJECT, a: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
    case a of
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ LONG ∪ ULONG ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪
                COMPOUNDATTRIBUTE ∪ PROTOTYPE ∪ PACKAGE ∪ GLOBAL do
            throw badValueError;
        CLASS do return a.call(this, args, phase);
        INSTANCE do
            Note that resolveAlias is not called when getting the env field.
            return resolveAlias(a).call(this, args, a.env, phase);
        METHODCLOSURE do
            code: INSTANCE ← a.method.code;
            return call(a.this, code, args, phase)
    end case
end proc;
```

```
proc construct(a: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
    case a of
        UNDEFINED | NULL | BOOLEAN | FLOAT64 | LONG | ULONG | CHARACTER | STRING | NAMESPACE |
            COMPOUNDATTRIBUTE | METHODCLOSURE | PROTOTYPE | PACKAGE | GLOBAL do
            throw badValueError;
        CLASS do return a.construct(args, phase);
        INSTANCE do
            Note that resolveAlias is not called when getting the env field.
            return resolveAlias(a).construct(args, a.env, phase)
    end case
end proc;
```

12.9 Member Operators

Syntax

```
MemberOperator ::
    DotOperator
  | . ParenExpression

DotOperator ::
    . QualifiedIdentifier
  | Brackets

Brackets ::
    [ ]
  | [ ListExpression^allowIn ]
  | [ NamedArgumentList ]

Arguments ::
    ParenExpressions
  | ( NamedArgumentList )

ParenExpressions ::
    ( )
  | ParenListExpression

NamedArgumentList ::
    LiteralField
  | ListExpression^allowIn , LiteralField
  | NamedArgumentList , LiteralField
```

Validation

```
proc Validate[MemberOperator] (cxt: CONTEXT, env: ENVIRONMENT)
    [MemberOperator :: DotOperator] do Validate[DotOperator](cxt, env);
    [MemberOperator :: . ParenExpression] do Validate[ParenExpression](cxt, env)
end proc;

proc Validate[DotOperator] (cxt: CONTEXT, env: ENVIRONMENT)
    [DotOperator :: . QualifiedIdentifier] do Validate[QualifiedIdentifier](cxt, env);
    [DotOperator :: Brackets] do Validate[Brackets](cxt, env)
end proc;
```

proc Validate[*Brackets*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*Brackets* ⇒ **[ ]**] **do nothing**;
[*Brackets* ⇒ **[** *ListExpression*^{allowIn} **]**] **do** Validate[*ListExpression*^{allowIn}](*cxt*, *env*);
[*Brackets* ⇒ **[** *NamedArgumentList* **]**] **do** Validate[*NamedArgumentList*](*cxt*, *env*)
end proc;

proc Validate[*Arguments*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*Arguments* ⇒ *ParenExpressions*] **do** Validate[*ParenExpressions*](*cxt*, *env*);
[*Arguments* ⇒ **(** *NamedArgumentList* **)**] **do** Validate[*NamedArgumentList*](*cxt*, *env*)
end proc;

proc Validate[*ParenExpressions*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ParenExpressions* ⇒ **( )**] **do nothing**;
[*ParenExpressions* ⇒ *ParenListExpression*] **do** Validate[*ParenListExpression*](*cxt*, *env*)
end proc;

proc Validate[*NamedArgumentList*] (*cxt*: CONTEXT, *env*: ENVIRONMENT): STRING{}
[*NamedArgumentList* ⇒ *LiteralField*] **do return** Validate[*LiteralField*](*cxt*, *env*);
[*NamedArgumentList* ⇒ *ListExpression*^{allowIn} **,** *LiteralField*] **do**
Validate[*ListExpression*^{allowIn}](*cxt*, *env*);
return Validate[*LiteralField*](*cxt*, *env*);
[*NamedArgumentList*$_0$ ⇒ *NamedArgumentList*$_1$ **,** *LiteralField*] **do**
names1: STRING{} ← Validate[*NamedArgumentList*$_1$](*cxt*, *env*);
names2: STRING{} ← Validate[*LiteralField*](*cxt*, *env*);
if *names1* ∪ *names2* ≠ {} **then throw syntaxError end if**;
return *names1* ∪ *names2*
end proc;

Evaluation

proc Eval[*MemberOperator*] (*env*: ENVIRONMENT, *base*: OBJECT, *phase*: PHASE): OBJORREF
[*MemberOperator* ⇒ *DotOperator*] **do return** Eval[*DotOperator*](*env*, *base*, *phase*);
[*MemberOperator* ⇒ **.** *ParenExpression*] **do** ????
end proc;

proc Eval[*DotOperator*] (*env*: ENVIRONMENT, *base*: OBJOPTIONALLIMIT, *phase*: PHASE): OBJORREF
[*DotOperator* ⇒ **.** *QualifiedIdentifier*] **do**
return DOTREFERENCE⟦base: *base*, propertyMultiname: Multiname[*QualifiedIdentifier*]⟧
[*DotOperator* ⇒ *Brackets*] **do**
args: ARGUMENTLIST ← Eval[*Brackets*](*env*, *phase*);
return BRACKETREFERENCE⟦base: *base*, args: *args*⟧
end proc;

proc Eval[*Brackets*] (*env*: ENVIRONMENT, *phase*: PHASE): ARGUMENTLIST
[*Brackets* ⇒ **[ ]**] **do return** ARGUMENTLIST⟦positional: [], named: {}⟧
[*Brackets* ⇒ **[** *ListExpression*^{allowIn} **]**] **do**
positional: OBJECT[] ← EvalAsList[*ListExpression*^{allowIn}](*env*, *phase*);
return ARGUMENTLIST⟦positional: *positional*, named: {}⟧
[*Brackets* ⇒ **[** *NamedArgumentList* **]**] **do return** Eval[*NamedArgumentList*](*env*, *phase*)
end proc;

```
proc Eval[Arguments] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
    [Arguments ⇒ ParenExpressions] do return Eval[ParenExpressions](env, phase);
    [Arguments ⇒ ( NamedArgumentList )] do return Eval[NamedArgumentList](env, phase)
end proc;

proc Eval[ParenExpressions] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
    [ParenExpressions ⇒ ( )] do return ARGUMENTLIST⟦positional: [], named: {}⟧;
    [ParenExpressions ⇒ ParenListExpression] do
        positional: OBJECT[] ← EvalAsList[ParenListExpression](env, phase);
        return ARGUMENTLIST⟦positional: positional, named: {}⟧
end proc;

proc Eval[NamedArgumentList] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
    [NamedArgumentList ⇒ LiteralField] do
        na: NAMEDARGUMENT ← Eval[LiteralField](env, phase);
        return ARGUMENTLIST⟦positional: [], named: {na}⟧;
    [NamedArgumentList ⇒ ListExpression^allowIn , LiteralField] do
        positional: OBJECT[] ← EvalAsList[ListExpression^allowIn](env, phase);
        na: NAMEDARGUMENT ← Eval[LiteralField](env, phase);
        return ARGUMENTLIST⟦positional: positional, named: {na}⟧;
    [NamedArgumentList_0 ⇒ NamedArgumentList_1 , LiteralField] do
        args: ARGUMENTLIST ← Eval[NamedArgumentList_1](env, phase);
        na: NAMEDARGUMENT ← Eval[LiteralField](env, phase);
        if some na2 ∈ args.named satisfies na2.name = na.name then
            throw argumentMismatchError
        end if;
        return ARGUMENTLIST⟦positional: args.positional, named: args.named ∪ {na}⟧
end proc;
```

12.10 Unary Operators

Syntax

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

Validation

```
Strict[UnaryExpression]: BOOLEAN;

proc Validate[UnaryExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [UnaryExpression ⇒ PostfixExpression] do Validate[PostfixExpression](cxt, env);
    [UnaryExpression ⇒ delete PostfixExpression] do
        Validate[PostfixExpression](cxt, env);
        Strict[UnaryExpression] ← cxt.strict;
```

```
    [UnaryExpression0 void UnaryExpression1] do Validate[UnaryExpression1](cxt, env);
    [UnaryExpression0 typeof UnaryExpression1] do
        Validate[UnaryExpression1](cxt, env);
    [UnaryExpression ++ PostfixExpression] do Validate[PostfixExpression](cxt, env);
    [UnaryExpression -- PostfixExpression] do Validate[PostfixExpression](cxt, env);
    [UnaryExpression0 + UnaryExpression1] do Validate[UnaryExpression1](cxt, env);
    [UnaryExpression0 - UnaryExpression1] do Validate[UnaryExpression1](cxt, env);
    [UnaryExpression0 ~ UnaryExpression1] do Validate[UnaryExpression1](cxt, env);
    [UnaryExpression0 ! UnaryExpression1] do Validate[UnaryExpression1](cxt, env)
end proc;
```

Evaluation

```
proc Eval[UnaryExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [UnaryExpression PostfixExpression] do return Eval[PostfixExpression](env, phase);
    [UnaryExpression delete PostfixExpression] do
        if phase = compile then throw compileExpressionError end if;
        r: OBJORREF Eval[PostfixExpression](env, phase);
        return deleteReference(r, Strict[UnaryExpression], phase);
    [UnaryExpression0 void UnaryExpression1] do
        r: OBJORREF Eval[UnaryExpression1](env, phase);
        readReference(r, phase);
        return undefined;
    [UnaryExpression0 typeof UnaryExpression1] do
        r: OBJORREF Eval[UnaryExpression1](env, phase);
        a: OBJECT readReference(r, phase);
        case a of
            UNDEFINED do return "undefined";
            NULL PROTOTYPE PACKAGE GLOBAL do return "object";
            BOOLEAN do return "boolean";
            FLOAT64 do return "number";
            LONG do return "long";
            ULONG do return "ulong";
            CHARACTER do return "character";
            STRING do return "string";
            NAMESPACE do return "namespace";
            COMPOUNDATTRIBUTE do return "attribute";
            CLASS METHODCLOSURE do return "function";
            INSTANCE do return resolveAlias(a).typeofString
        end case;
    [UnaryExpression ++ PostfixExpression] do
        if phase = compile then throw compileExpressionError end if;
        r: OBJORREF Eval[PostfixExpression](env, phase);
        a: OBJECT readReference(r, phase);
        b: OBJECT plus(a, phase);
        c: OBJECT add(b, 1.0, phase);
        writeReference(r, c, phase);
        return c;
```

```
    [UnaryExpression □ -- PostfixExpression] do
        if phase = compile then throw compileExpressionError end if;
        r: OBJORREF □ Eval[PostfixExpression](env, phase);
        a: OBJECT □ readReference(r, phase);
        b: OBJECT □ plus(a, phase);
        c: OBJECT □ subtract(b, 1.0, phase);
        writeReference(r, c, phase);
        return c;
    [UnaryExpression0 □ + UnaryExpression1] do
        r: OBJORREF □ Eval[UnaryExpression1](env, phase);
        a: OBJECT □ readReference(r, phase);
        return plus(a, phase);
    [UnaryExpression0 □ - UnaryExpression1] do
        r: OBJORREF □ Eval[UnaryExpression1](env, phase);
        a: OBJECT □ readReference(r, phase);
        return minus(a, phase);
    [UnaryExpression0 □ ~ UnaryExpression1] do
        r: OBJORREF □ Eval[UnaryExpression1](env, phase);
        a: OBJECT □ readReference(r, phase);
        return bitNot(a, phase);
    [UnaryExpression0 □ ! UnaryExpression1] do
        r: OBJORREF □ Eval[UnaryExpression1](env, phase);
        a: OBJECT □ readReference(r, phase);
        return logicalNot(a, phase)
end proc;
```

plus(*a*, *phase*) returns the value of the unary expression +*a*. If *phase* is **compile**, only compile-time operations are permitted.

```
proc plus(a: OBJECT, phase: PHASE): OBJECT
    return toGeneralNumber(a, phase)
end proc;

proc minus(a: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER □ toGeneralNumber(a, phase);
    case x of
        FLOAT64 do return float64Negate(x);
        LONG □ ULONG do i: INTEGER □ –(x.value); return checkLong(i)
    end case
end proc;

proc bitNot(a: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER □ toGeneralNumber(a, phase);
    case x of
        FLOAT64 do
            i: {–2^31 ... 2^31 – 1} □ signedWrap32(truncateToInteger(x));
            return realToFloat64(bitwiseXor(i, –1));
        LONG do i: {–2^63 ... 2^63 – 1} □ x.value; return LONG□value: bitwiseXor(i, –1)□
        ULONG do
            i: {0 ... 2^64 – 1} □ x.value;
            return ULONG□value: bitwiseXor(i, 0xFFFFFFFFFFFFFFFF)□
    end case
end proc;
```

logicalNot(*a*, *phase*) returns the value of the unary expression !*a*. If *phase* is **compile**, only compile-time operations are permitted.

```
proc logicalNot(a: OBJECT, phase: PHASE): OBJECT
    return not toBoolean(a, phase)
end proc;
```

12.11 Multiplicative Operators

Syntax

```
MultiplicativeExpression ⊐
    UnaryExpression
  | MultiplicativeExpression * UnaryExpression
  | MultiplicativeExpression / UnaryExpression
  | MultiplicativeExpression % UnaryExpression
```

Validation

```
proc Validate[MultiplicativeExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [MultiplicativeExpression ⊐ UnaryExpression] do Validate[UnaryExpression](cxt, env);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 * UnaryExpression] do
        Validate[MultiplicativeExpression1](cxt, env);
        Validate[UnaryExpression](cxt, env);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 / UnaryExpression] do
        Validate[MultiplicativeExpression1](cxt, env);
        Validate[UnaryExpression](cxt, env);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 % UnaryExpression] do
        Validate[MultiplicativeExpression1](cxt, env);
        Validate[UnaryExpression](cxt, env)
end proc;
```

Evaluation

```
proc Eval[MultiplicativeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [MultiplicativeExpression ⊐ UnaryExpression] do
        return Eval[UnaryExpression](env, phase);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 * UnaryExpression] do
        ra: OBJORREF ⊐ Eval[MultiplicativeExpression1](env, phase);
        a: OBJECT ⊐ readReference(ra, phase);
        rb: OBJORREF ⊐ Eval[UnaryExpression](env, phase);
        b: OBJECT ⊐ readReference(rb, phase);
        return multiply(a, b, phase);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 / UnaryExpression] do
        ra: OBJORREF ⊐ Eval[MultiplicativeExpression1](env, phase);
        a: OBJECT ⊐ readReference(ra, phase);
        rb: OBJORREF ⊐ Eval[UnaryExpression](env, phase);
        b: OBJECT ⊐ readReference(rb, phase);
        return divide(a, b, phase);
    [MultiplicativeExpression0 ⊐ MultiplicativeExpression1 % UnaryExpression] do
        ra: OBJORREF ⊐ Eval[MultiplicativeExpression1](env, phase);
        a: OBJECT ⊐ readReference(ra, phase);
        rb: OBJORREF ⊐ Eval[UnaryExpression](env, phase);
        b: OBJECT ⊐ readReference(rb, phase);
        return remainder(a, b, phase)
end proc;
```

```
proc multiply(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    y: GENERALNUMBER ← toGeneralNumber(b, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: INTEGER ← checkInteger(x);
        j: INTEGER ← checkInteger(y);
        k: INTEGER ← i×j;
        if x ∈ ULONG or y ∈ ULONG then return checkULong(k)
        else return checkLong(k)
        end if
    else return float64Multiply(x, y)
    end if
end proc;

proc divide(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    y: GENERALNUMBER ← toGeneralNumber(b, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: INTEGER ← checkInteger(x);
        j: INTEGER ← checkInteger(y);
        if j = 0 then throw rangeError end if;
        q: RATIONAL ← i/j;
        k: INTEGER ← q ≥ 0 ? ⌊q⌋ : ⌈q⌉;
        if x ∈ ULONG or y ∈ ULONG then return checkULong(k)
        else return checkLong(k)
        end if
    else return float64Divide(x, y)
    end if
end proc;

proc remainder(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    y: GENERALNUMBER ← toGeneralNumber(b, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: INTEGER ← checkInteger(x);
        j: INTEGER ← checkInteger(y);
        if j = 0 then throw rangeError end if;
        q: RATIONAL ← i/j;
        k: INTEGER ← q ≥ 0 ? ⌊q⌋ : ⌈q⌉;
        r: INTEGER ← i − j×k;
        if x ∈ ULONG then return ULONG{value: r} else return LONG{value: r} end if
    else return float64Remainder(x, y)
    end if
end proc;
```

12.12 Additive Operators

Syntax

```
AdditiveExpression ::
    MultiplicativeExpression
  | AdditiveExpression + MultiplicativeExpression
  | AdditiveExpression − MultiplicativeExpression
```

Validation

```
proc Validate[AdditiveExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [AdditiveExpression ⇒ MultiplicativeExpression] do
        Validate[MultiplicativeExpression](cxt, env);
    [AdditiveExpression0 ⇒ AdditiveExpression1 + MultiplicativeExpression] do
        Validate[AdditiveExpression1](cxt, env);
        Validate[MultiplicativeExpression](cxt, env);
    [AdditiveExpression0 ⇒ AdditiveExpression1 − MultiplicativeExpression] do
        Validate[AdditiveExpression1](cxt, env);
        Validate[MultiplicativeExpression](cxt, env)
end proc;
```

Evaluation

```
proc Eval[AdditiveExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [AdditiveExpression ⇒ MultiplicativeExpression] do
        return Eval[MultiplicativeExpression](env, phase);
    [AdditiveExpression0 ⇒ AdditiveExpression1 + MultiplicativeExpression] do
        ra: OBJORREF ⇐ Eval[AdditiveExpression1](env, phase);
        a: OBJECT ⇐ readReference(ra, phase);
        rb: OBJORREF ⇐ Eval[MultiplicativeExpression](env, phase);
        b: OBJECT ⇐ readReference(rb, phase);
        return add(a, b, phase);
    [AdditiveExpression0 ⇒ AdditiveExpression1 − MultiplicativeExpression] do
        ra: OBJORREF ⇐ Eval[AdditiveExpression1](env, phase);
        a: OBJECT ⇐ readReference(ra, phase);
        rb: OBJORREF ⇐ Eval[MultiplicativeExpression](env, phase);
        b: OBJECT ⇐ readReference(rb, phase);
        return subtract(a, b, phase)
end proc;

proc add(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    ap: PRIMITIVEOBJECT ⇐ toPrimitive(a, null, phase);
    bp: PRIMITIVEOBJECT ⇐ toPrimitive(b, null, phase);
    if ap ∈ CHARACTER ∪ STRING or bp ∈ CHARACTER ∪ STRING then
        return toString(ap, phase) ⊕ toString(bp, phase)
    end if;
    x: GENERALNUMBER ⇐ toGeneralNumber(ap, phase);
    y: GENERALNUMBER ⇐ toGeneralNumber(bp, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: INTEGER ⇐ checkInteger(x);
        j: INTEGER ⇐ checkInteger(y);
        k: INTEGER ⇐ i + j;
        if x ∈ ULONG or y ∈ ULONG then return checkULong(k)
        else return checkLong(k)
        end if
    else return float64Add(x, y)
    end if
end proc;
```

```
proc subtract(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    y: GENERALNUMBER ← toGeneralNumber(b, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: INTEGER ← checkInteger(x);
        j: INTEGER ← checkInteger(y);
        k: INTEGER ← i − j;
        if x ∈ ULONG then return checkULong(k) else return checkLong(k) end if
    else return float64Subtract(x, y)
    end if
end proc;
```

12.13 Bitwise Shift Operators

Syntax

```
ShiftExpression ::
      AdditiveExpression
    | ShiftExpression << AdditiveExpression
    | ShiftExpression >> AdditiveExpression
    | ShiftExpression >>> AdditiveExpression
```

Validation

```
proc Validate[ShiftExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [ShiftExpression ::= AdditiveExpression] do Validate[AdditiveExpression](cxt, env);
    [ShiftExpression0 ::= ShiftExpression1 << AdditiveExpression] do
        Validate[ShiftExpression1](cxt, env);
        Validate[AdditiveExpression](cxt, env);
    [ShiftExpression0 ::= ShiftExpression1 >> AdditiveExpression] do
        Validate[ShiftExpression1](cxt, env);
        Validate[AdditiveExpression](cxt, env);
    [ShiftExpression0 ::= ShiftExpression1 >>> AdditiveExpression] do
        Validate[ShiftExpression1](cxt, env);
        Validate[AdditiveExpression](cxt, env)
end proc;
```

Evaluation

```
proc Eval[ShiftExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ShiftExpression ::= AdditiveExpression] do
        return Eval[AdditiveExpression](env, phase);
    [ShiftExpression0 ::= ShiftExpression1 << AdditiveExpression] do
        ra: OBJORREF ← Eval[ShiftExpression1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[AdditiveExpression](env, phase);
        b: OBJECT ← readReference(rb, phase);
        return shiftLeft(a, b, phase);
    [ShiftExpression0 ::= ShiftExpression1 >> AdditiveExpression] do
        ra: OBJORREF ← Eval[ShiftExpression1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[AdditiveExpression](env, phase);
        b: OBJECT ← readReference(rb, phase);
        return shiftRight(a, b, phase);
```

```
    [ShiftExpression0 ← ShiftExpression1 >>> AdditiveExpression] do
        ra: OBJORREF ← Eval[ShiftExpression1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[AdditiveExpression](env, phase);
        b: OBJECT ← readReference(rb, phase);
        return shiftRightUnsigned(a, b, phase)
end proc;

proc shiftLeft(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    count: INTEGER ← truncateToInteger(toGeneralNumber(b, phase));
    case x of
        FLOAT64 do
            i: {–2^31 ... 2^31 – 1} ← signedWrap32(truncateToInteger(x));
            count ← bitwiseAnd(count, 0x1F);
            i ← signedWrap32(bitwiseShift(i, count));
            return realToFloat64(i);
        LONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {–2^63 ... 2^63 – 1} ← signedWrap64(bitwiseShift(x.value, count));
            return LONG[value: i]
        ULONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {0 ... 2^64 – 1} ← unsignedWrap64(bitwiseShift(x.value, count));
            return ULONG[value: i]
    end case
end proc;

proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    count: INTEGER ← truncateToInteger(toGeneralNumber(b, phase));
    case x of
        FLOAT64 do
            i: {–2^31 ... 2^31 – 1} ← signedWrap32(truncateToInteger(x));
            count ← bitwiseAnd(count, 0x1F);
            i ← bitwiseShift(i, –count);
            return realToFloat64(i);
        LONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {–2^63 ... 2^63 – 1} ← bitwiseShift(x.value, –count);
            return LONG[value: i]
        ULONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {–2^63 ... 2^63 – 1} ← bitwiseShift(signedWrap64(x.value), –count);
            return ULONG[value: unsignedWrap64(i)]
    end case
end proc;
```

```
proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    count: INTEGER ← truncateToInteger(toGeneralNumber(b, phase));
    case x of
        FLOAT64 do
            i: {0 ... 2^32 – 1} ← unsignedWrap32(truncateToInteger(x));
            count ← bitwiseAnd(count, 0x1F);
            i ← bitwiseShift(i, –count);
            return realToFloat64(i);
        LONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {0 ... 2^64 – 1} ← bitwiseShift(unsignedWrap64(x.value), –count);
            return LONG[value: signedWrap64(i)]
        ULONG do
            count ← bitwiseAnd(count, 0x3F);
            i: {0 ... 2^64 – 1} ← bitwiseShift(x.value, –count);
            return ULONG[value: i]
    end case
end proc;
```

12.14 Relational Operators

Syntax

```
RelationalExpression^allowIn ≡
    ShiftExpression
  | RelationalExpression^allowIn < ShiftExpression
  | RelationalExpression^allowIn > ShiftExpression
  | RelationalExpression^allowIn <= ShiftExpression
  | RelationalExpression^allowIn >= ShiftExpression
  | RelationalExpression^allowIn is ShiftExpression
  | RelationalExpression^allowIn as ShiftExpression
  | RelationalExpression^allowIn in ShiftExpression
  | RelationalExpression^allowIn instanceof ShiftExpression

RelationalExpression^noIn ≡
    ShiftExpression
  | RelationalExpression^noIn < ShiftExpression
  | RelationalExpression^noIn > ShiftExpression
  | RelationalExpression^noIn <= ShiftExpression
  | RelationalExpression^noIn >= ShiftExpression
  | RelationalExpression^noIn is ShiftExpression
  | RelationalExpression^noIn as ShiftExpression
  | RelationalExpression^noIn instanceof ShiftExpression
```

Validation

```
proc Validate[RelationalExpression^α] (cxt: CONTEXT, env: ENVIRONMENT)
    [RelationalExpression^α ≡ ShiftExpression] do Validate[ShiftExpression](cxt, env);
    [RelationalExpression^α0 ≡ RelationalExpression^α1 < ShiftExpression] do
        Validate[RelationalExpression^α1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression^α0 ≡ RelationalExpression^α1 > ShiftExpression] do
        Validate[RelationalExpression^α1](cxt, env);
        Validate[ShiftExpression](cxt, env);
```

```
    [RelationalExpression□0 □ RelationalExpression□1 <= ShiftExpression] do
        Validate[RelationalExpression□1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression□0 □ RelationalExpression□1 >= ShiftExpression] do
        Validate[RelationalExpression□1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression□0 □ RelationalExpression□1 is ShiftExpression] do
        Validate[RelationalExpression□1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression□0 □ RelationalExpression□1 as ShiftExpression] do
        Validate[RelationalExpression□1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression^allowIn_0 □ RelationalExpression^allowIn_1 in ShiftExpression] do
        Validate[RelationalExpression^allowIn_1](cxt, env);
        Validate[ShiftExpression](cxt, env);
    [RelationalExpression□0 □ RelationalExpression□1 instanceof ShiftExpression] do
        Validate[RelationalExpression□1](cxt, env);
        Validate[ShiftExpression](cxt, env)
end proc;
```

Evaluation

```
proc Eval[RelationalExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [RelationalExpression□ □ ShiftExpression] do
        return Eval[ShiftExpression](env, phase);
    [RelationalExpression□0 □ RelationalExpression□1 < ShiftExpression] do
        ra: OBJORREF □ Eval[RelationalExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[ShiftExpression](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return isLess(a, b, phase);
    [RelationalExpression□0 □ RelationalExpression□1 > ShiftExpression] do
        ra: OBJORREF □ Eval[RelationalExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[ShiftExpression](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return isLess(b, a, phase);
    [RelationalExpression□0 □ RelationalExpression□1 <= ShiftExpression] do
        ra: OBJORREF □ Eval[RelationalExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[ShiftExpression](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return isLessOrEqual(a, b, phase);
    [RelationalExpression□0 □ RelationalExpression□1 >= ShiftExpression] do
        ra: OBJORREF □ Eval[RelationalExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[ShiftExpression](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return isLessOrEqual(b, a, phase);
```

```
    [RelationalExpression□ □ RelationalExpression□ is ShiftExpression] do ????;
    [RelationalExpression□ □ RelationalExpression□ as ShiftExpression] do ????;
    [RelationalExpression^allowIn □ RelationalExpression^allowIn in ShiftExpression] do
        ????;
    [RelationalExpression□ □ RelationalExpression□ instanceof ShiftExpression] do ????
end proc;

proc isLess(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    ap: PRIMITIVEOBJECT □ toPrimitive(a, null, phase);
    bp: PRIMITIVEOBJECT □ toPrimitive(b, null, phase);
    if ap □ CHARACTER □ STRING and bp □ CHARACTER □ STRING then
        return toString(ap, phase) < toString(bp, phase)
    end if;
    return generalNumberCompare(toGeneralNumber(ap, phase), toGeneralNumber(bp, phase)) = less
end proc;

proc isLessOrEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    ap: PRIMITIVEOBJECT □ toPrimitive(a, null, phase);
    bp: PRIMITIVEOBJECT □ toPrimitive(b, null, phase);
    if ap □ CHARACTER □ STRING and bp □ CHARACTER □ STRING then
        return toString(ap, phase) ≤ toString(bp, phase)
    end if;
    return generalNumberCompare(toGeneralNumber(ap, phase), toGeneralNumber(bp, phase)) □ {less, equal}
end proc;
```

12.15 Equality Operators

Syntax

```
EqualityExpression□ □
    RelationalExpression□
  | EqualityExpression□ == RelationalExpression□
  | EqualityExpression□ != RelationalExpression□
  | EqualityExpression□ === RelationalExpression□
  | EqualityExpression□ !== RelationalExpression□
```

Validation

```
proc Validate[EqualityExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    [EqualityExpression□ □ RelationalExpression□] do
        Validate[RelationalExpression□](cxt, env);
    [EqualityExpression□0 □ EqualityExpression□1 == RelationalExpression□] do
        Validate[EqualityExpression□1](cxt, env);
        Validate[RelationalExpression□](cxt, env);
    [EqualityExpression□0 □ EqualityExpression□1 != RelationalExpression□] do
        Validate[EqualityExpression□1](cxt, env);
        Validate[RelationalExpression□](cxt, env);
    [EqualityExpression□0 □ EqualityExpression□1 === RelationalExpression□] do
        Validate[EqualityExpression□1](cxt, env);
        Validate[RelationalExpression□](cxt, env);
    [EqualityExpression□0 □ EqualityExpression□1 !== RelationalExpression□] do
        Validate[EqualityExpression□1](cxt, env);
        Validate[RelationalExpression□](cxt, env)
end proc;
```

Evaluation

```
proc Eval[EqualityExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [EqualityExpression□ ⇒ RelationalExpression□] do
        return Eval[RelationalExpression□](env, phase);
    [EqualityExpression□0 ⇒ EqualityExpression□1 == RelationalExpression□] do
        ra: OBJORREF ← Eval[EqualityExpression□1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[RelationalExpression□](env, phase);
        b: OBJECT ← readReference(rb, phase);
        return isEqual(a, b, phase);
    [EqualityExpression□0 ⇒ EqualityExpression□1 != RelationalExpression□] do
        ra: OBJORREF ← Eval[EqualityExpression□1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[RelationalExpression□](env, phase);
        b: OBJECT ← readReference(rb, phase);
        c: BOOLEAN ← isEqual(a, b, phase);
        return not c;
    [EqualityExpression□0 ⇒ EqualityExpression□1 === RelationalExpression□] do
        ra: OBJORREF ← Eval[EqualityExpression□1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[RelationalExpression□](env, phase);
        b: OBJECT ← readReference(rb, phase);
        return isStrictlyEqual(a, b, phase);
    [EqualityExpression□0 ⇒ EqualityExpression□1 !== RelationalExpression□] do
        ra: OBJORREF ← Eval[EqualityExpression□1](env, phase);
        a: OBJECT ← readReference(ra, phase);
        rb: OBJORREF ← Eval[RelationalExpression□](env, phase);
        b: OBJECT ← readReference(rb, phase);
        c: BOOLEAN ← isStrictlyEqual(a, b, phase);
        return not c
end proc;
```

```
proc isEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    case a of
        UNDEFINED ∪ NULL do return b ∈ UNDEFINED ∪ NULL;
        BOOLEAN do
            if b ∈ BOOLEAN then return a = b
            else return isEqual(toGeneralNumber(a, phase), b, phase)
            end if;
        GENERALNUMBER do
            bp: PRIMITIVEOBJECT ∪ toPrimitive(b, null, phase);
            case bp of
                UNDEFINED ∪ NULL do return false;
                BOOLEAN ∪ GENERALNUMBER ∪ CHARACTER ∪ STRING do
                    return generalNumberCompare(a, toGeneralNumber(bp, phase)) = equal
            end case;
        CHARACTER ∪ STRING do
            bp: PRIMITIVEOBJECT ∪ toPrimitive(b, null, phase);
            case bp of
                UNDEFINED ∪ NULL do return false;
                BOOLEAN ∪ GENERALNUMBER do
                    return generalNumberCompare(toGeneralNumber(a, phase), toGeneralNumber(bp, phase)) = equal;
                CHARACTER ∪ STRING do return toString(a, phase) = toString(bp, phase)
            end case;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE ∪ PACKAGE ∪
                GLOBAL do
            case b of
                UNDEFINED ∪ NULL do return false;
                NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE ∪
                        PACKAGE ∪ GLOBAL do
                    return isStrictlyEqual(a, b, phase);
                BOOLEAN ∪ GENERALNUMBER ∪ CHARACTER ∪ STRING do
                    ap: PRIMITIVEOBJECT ∪ toPrimitive(a, null, phase);
                    return isEqual(ap, b, phase)
            end case
    end case
end proc;

proc isStrictlyEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    if a ∈ ALIASINSTANCE then return isStrictlyEqual(a.original, b, phase)
    elsif b ∈ ALIASINSTANCE then return isStrictlyEqual(a, b.original, phase)
    elsif a ∈ FLOAT64 and b ∈ FLOAT64 then return float64Compare(a, b) = equal
    else
        Note that a LONG 5 is strictly equal to itself but not to ULONG 5 or FLOAT64 5.
        return a = b
    end if
end proc;
```

12.16 Binary Bit Operators

Syntax

```
BitwiseAndExpression[] ∪
    EqualityExpression[]
  | BitwiseAndExpression[] & EqualityExpression[]
```

BitwiseXorExpression□ □
BitwiseAndExpression□
| *BitwiseXorExpression*□ ^ *BitwiseAndExpression*□

BitwiseOrExpression□ □
BitwiseXorExpression□
| *BitwiseOrExpression*□ | *BitwiseXorExpression*□

Validation

proc Validate[*BitwiseAndExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*BitwiseAndExpression*□ □ *EqualityExpression*□] **do**
Validate[*EqualityExpression*□](*cxt*, *env*);
[*BitwiseAndExpression*□$_0$ □ *BitwiseAndExpression*□$_1$ & *EqualityExpression*□] **do**
Validate[*BitwiseAndExpression*□$_1$](*cxt*, *env*);
Validate[*EqualityExpression*□](*cxt*, *env*)
end proc;

proc Validate[*BitwiseXorExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*BitwiseXorExpression*□ □ *BitwiseAndExpression*□] **do**
Validate[*BitwiseAndExpression*□](*cxt*, *env*);
[*BitwiseXorExpression*□$_0$ □ *BitwiseXorExpression*□$_1$ ^ *BitwiseAndExpression*□] **do**
Validate[*BitwiseXorExpression*□$_1$](*cxt*, *env*);
Validate[*BitwiseAndExpression*□](*cxt*, *env*)
end proc;

proc Validate[*BitwiseOrExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*BitwiseOrExpression*□ □ *BitwiseXorExpression*□] **do**
Validate[*BitwiseXorExpression*□](*cxt*, *env*);
[*BitwiseOrExpression*□$_0$ □ *BitwiseOrExpression*□$_1$ | *BitwiseXorExpression*□] **do**
Validate[*BitwiseOrExpression*□$_1$](*cxt*, *env*);
Validate[*BitwiseXorExpression*□](*cxt*, *env*)
end proc;

Evaluation

proc Eval[*BitwiseAndExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*BitwiseAndExpression*□ □ *EqualityExpression*□] **do**
return Eval[*EqualityExpression*□](*env*, *phase*);
[*BitwiseAndExpression*□$_0$ □ *BitwiseAndExpression*□$_1$ & *EqualityExpression*□] **do**
ra: OBJORREF □ Eval[*BitwiseAndExpression*□$_1$](*env*, *phase*);
a: OBJECT □ *readReference*(*ra*, *phase*);
rb: OBJORREF □ Eval[*EqualityExpression*□](*env*, *phase*);
b: OBJECT □ *readReference*(*rb*, *phase*);
return *bitAnd*(*a*, *b*, *phase*)
end proc;

proc Eval[*BitwiseXorExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*BitwiseXorExpression*□ □ *BitwiseAndExpression*□] **do**
return Eval[*BitwiseAndExpression*□](*env*, *phase*);

```
    [BitwiseXorExpression□0 □ BitwiseXorExpression□1 ^ BitwiseAndExpression□] do
        ra: OBJORREF □ Eval[BitwiseXorExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[BitwiseAndExpression□](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return bitXor(a, b, phase)
end proc;

proc Eval[BitwiseOrExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [BitwiseOrExpression□ □ BitwiseXorExpression□] do
        return Eval[BitwiseXorExpression□](env, phase);
    [BitwiseOrExpression□0 □ BitwiseOrExpression□1 | BitwiseXorExpression□] do
        ra: OBJORREF □ Eval[BitwiseOrExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[BitwiseXorExpression□](env, phase);
        b: OBJECT □ readReference(rb, phase);
        return bitOr(a, b, phase)
end proc;

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
    x: GENERALNUMBER □ toGeneralNumber(a, phase);
    y: GENERALNUMBER □ toGeneralNumber(b, phase);
    if x □ LONG □ ULONG or y □ LONG □ ULONG then
        i: {–2^63 ... 2^63 – 1} □ signedWrap64(checkInteger(x));
        j: {–2^63 ... 2^63 – 1} □ signedWrap64(checkInteger(y));
        k: {–2^63 ... 2^63 – 1} □ bitwiseAnd(i, j);
        if x □ ULONG or y □ ULONG then return ULONG□value: unsignedWrap64(k)□
        else return LONG□value: k□
        end if
    else
        i: INTEGER □ signedWrap32(truncateToInteger(x));
        j: INTEGER □ signedWrap32(truncateToInteger(y));
        return realToFloat64(bitwiseAnd(i, j))
    end if
end proc;

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
    x: GENERALNUMBER □ toGeneralNumber(a, phase);
    y: GENERALNUMBER □ toGeneralNumber(b, phase);
    if x □ LONG □ ULONG or y □ LONG □ ULONG then
        i: {–2^63 ... 2^63 – 1} □ signedWrap64(checkInteger(x));
        j: {–2^63 ... 2^63 – 1} □ signedWrap64(checkInteger(y));
        k: {–2^63 ... 2^63 – 1} □ bitwiseXor(i, j);
        if x □ ULONG or y □ ULONG then return ULONG□value: unsignedWrap64(k)□
        else return LONG□value: k□
        end if
    else
        i: INTEGER □ signedWrap32(truncateToInteger(x));
        j: INTEGER □ signedWrap32(truncateToInteger(y));
        return realToFloat64(bitwiseXor(i, j))
    end if
end proc;
```

```
proc bitOr(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
    x: GENERALNUMBER ← toGeneralNumber(a, phase);
    y: GENERALNUMBER ← toGeneralNumber(b, phase);
    if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
        i: {–2^63 ... 2^63 – 1} ← signedWrap64(checkInteger(x));
        j: {–2^63 ... 2^63 – 1} ← signedWrap64(checkInteger(y));
        k: {–2^63 ... 2^63 – 1} ← bitwiseOr(i, j);
        if x ∈ ULONG or y ∈ ULONG then return ULONG{value: unsignedWrap64(k)}
        else return LONG{value: k}
        end if
    else
        i: INTEGER ← signedWrap32(truncateToInteger(x));
        j: INTEGER ← signedWrap32(truncateToInteger(y));
        return realToFloat64(bitwiseOr(i, j))
    end if
end proc;
```

12.17 Binary Logical Operators

Syntax

```
LogicalAndExpression□ ::
    BitwiseOrExpression□
  | LogicalAndExpression□ && BitwiseOrExpression□

LogicalXorExpression□ ::
    LogicalAndExpression□
  | LogicalXorExpression□ ^^ LogicalAndExpression□

LogicalOrExpression□ ::
    LogicalXorExpression□
  | LogicalOrExpression□ || LogicalXorExpression□
```

Validation

```
proc Validate[LogicalAndExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    [LogicalAndExpression□ ::  BitwiseOrExpression□] do
        Validate[BitwiseOrExpression□](cxt, env);
    [LogicalAndExpression□_0 ::  LogicalAndExpression□_1 && BitwiseOrExpression□] do
        Validate[LogicalAndExpression□_1](cxt, env);
        Validate[BitwiseOrExpression□](cxt, env)
end proc;

proc Validate[LogicalXorExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    [LogicalXorExpression□ ::  LogicalAndExpression□] do
        Validate[LogicalAndExpression□](cxt, env);
    [LogicalXorExpression□_0 ::  LogicalXorExpression□_1 ^^ LogicalAndExpression□] do
        Validate[LogicalXorExpression□_1](cxt, env);
        Validate[LogicalAndExpression□](cxt, env)
end proc;

proc Validate[LogicalOrExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    [LogicalOrExpression□ ::  LogicalXorExpression□] do
        Validate[LogicalXorExpression□](cxt, env);
```

```
    [LogicalOrExpression□0 □ LogicalOrExpression□1 || LogicalXorExpression□] do
        Validate[LogicalOrExpression□1](cxt, env);
        Validate[LogicalXorExpression□](cxt, env)
end proc;
```

Evaluation

```
proc Eval[LogicalAndExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [LogicalAndExpression□ □ BitwiseOrExpression□] do
        return Eval[BitwiseOrExpression□](env, phase);
    [LogicalAndExpression□0 □ LogicalAndExpression□1 && BitwiseOrExpression□] do
        ra: OBJORREF □ Eval[LogicalAndExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        if toBoolean(a, phase) then
            rb: OBJORREF □ Eval[BitwiseOrExpression□](env, phase);
            return readReference(rb, phase)
        else return a
        end if
end proc;

proc Eval[LogicalXorExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [LogicalXorExpression□ □ LogicalAndExpression□] do
        return Eval[LogicalAndExpression□](env, phase);
    [LogicalXorExpression□0 □ LogicalXorExpression□1 ^^ LogicalAndExpression□] do
        ra: OBJORREF □ Eval[LogicalXorExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        rb: OBJORREF □ Eval[LogicalAndExpression□](env, phase);
        b: OBJECT □ readReference(rb, phase);
        ba: BOOLEAN □ toBoolean(a, phase);
        bb: BOOLEAN □ toBoolean(b, phase);
        return ba xor bb
end proc;

proc Eval[LogicalOrExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [LogicalOrExpression□ □ LogicalXorExpression□] do
        return Eval[LogicalXorExpression□](env, phase);
    [LogicalOrExpression□0 □ LogicalOrExpression□1 || LogicalXorExpression□] do
        ra: OBJORREF □ Eval[LogicalOrExpression□1](env, phase);
        a: OBJECT □ readReference(ra, phase);
        if toBoolean(a, phase) then return a
        else
            rb: OBJORREF □ Eval[LogicalXorExpression□](env, phase);
            return readReference(rb, phase)
        end if
end proc;
```

12.18 Conditional Operator

Syntax

```
ConditionalExpression□ □
    LogicalOrExpression□
  | LogicalOrExpression□ ? AssignmentExpression□ : AssignmentExpression□
```

NonAssignmentExpression□ □
LogicalOrExpression□
| *LogicalOrExpression*□ **?** *NonAssignmentExpression*□ **:** *NonAssignmentExpression*□

Validation

proc Validate[*ConditionalExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ConditionalExpression*□ □ *LogicalOrExpression*□] **do**
Validate[*LogicalOrExpression*□](*cxt*, *env*);
[*ConditionalExpression*□ □ *LogicalOrExpression*□ **?** *AssignmentExpression*□$_1$ **:** *AssignmentExpression*□$_2$] **do**
Validate[*LogicalOrExpression*□](*cxt*, *env*);
Validate[*AssignmentExpression*□$_1$](*cxt*, *env*);
Validate[*AssignmentExpression*□$_2$](*cxt*, *env*)
end proc;

proc Validate[*NonAssignmentExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*NonAssignmentExpression*□ □ *LogicalOrExpression*□] **do**
Validate[*LogicalOrExpression*□](*cxt*, *env*);
[*NonAssignmentExpression*□$_0$ □ *LogicalOrExpression*□ **?** *NonAssignmentExpression*□$_1$ **:** *NonAssignmentExpression*□$_2$] **do**
Validate[*LogicalOrExpression*□](*cxt*, *env*);
Validate[*NonAssignmentExpression*□$_1$](*cxt*, *env*);
Validate[*NonAssignmentExpression*□$_2$](*cxt*, *env*)
end proc;

Evaluation

proc Eval[*ConditionalExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*ConditionalExpression*□ □ *LogicalOrExpression*□] **do**
return Eval[*LogicalOrExpression*□](*env*, *phase*);
[*ConditionalExpression*□ □ *LogicalOrExpression*□ **?** *AssignmentExpression*□$_1$ **:** *AssignmentExpression*□$_2$] **do**
ra: OBJORREF □ Eval[*LogicalOrExpression*□](*env*, *phase*);
a: OBJECT □ *readReference*(*ra*, *phase*);
if *toBoolean*(*a*, *phase*) **then**
rb: OBJORREF □ Eval[*AssignmentExpression*□$_1$](*env*, *phase*);
return *readReference*(*rb*, *phase*)
else
rc: OBJORREF □ Eval[*AssignmentExpression*□$_2$](*env*, *phase*);
return *readReference*(*rc*, *phase*)
end if
end proc;

proc Eval[*NonAssignmentExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*NonAssignmentExpression*□ □ *LogicalOrExpression*□] **do**
return Eval[*LogicalOrExpression*□](*env*, *phase*);

```
    [NonAssignmentExpression□0 □ LogicalOrExpression□ ? NonAssignmentExpression□1 : NonAssignmentExpression□
        2] do
      ra: OBJORREF □ Eval[LogicalOrExpression□](env, phase);
      a: OBJECT □ readReference(ra, phase);
      if toBoolean(a, phase) then
        rb: OBJORREF □ Eval[NonAssignmentExpression□1](env, phase);
        return readReference(rb, phase)
      else
        rc: OBJORREF □ Eval[NonAssignmentExpression□2](env, phase);
        return readReference(rc, phase)
      end if
  end proc;
```

12.19 Assignment Operators

Syntax

```
AssignmentExpression□ □
    ConditionalExpression□
  | PostfixExpression = AssignmentExpression□
  | PostfixExpression CompoundAssignment AssignmentExpression□
  | PostfixExpression LogicalAssignment AssignmentExpression□

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

LogicalAssignment □
    &&=
  | ^^=
  | ||=
```

Semantics

```
tag andEq;

tag xorEq;

tag orEq;
```

Validation

```
proc Validate[AssignmentExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    [AssignmentExpression□ □ ConditionalExpression□] do
      Validate[ConditionalExpression□](cxt, env);
```

```
    [AssignmentExpression□0 □ PostfixExpression = AssignmentExpression□1] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression□1](cxt, env);
    [AssignmentExpression□0 □ PostfixExpression CompoundAssignment AssignmentExpression□1] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression□1](cxt, env);
    [AssignmentExpression□0 □ PostfixExpression LogicalAssignment AssignmentExpression□1] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression□1](cxt, env)
end proc;
```

Evaluation

```
proc Eval[AssignmentExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [AssignmentExpression□ □ ConditionalExpression□] do
        return Eval[ConditionalExpression□](env, phase);
    [AssignmentExpression□0 □ PostfixExpression = AssignmentExpression□1] do
        if phase = compile then throw compileExpressionError end if;
        ra: OBJORREF □ Eval[PostfixExpression](env, phase);
        rb: OBJORREF □ Eval[AssignmentExpression□1](env, phase);
        b: OBJECT □ readReference(rb, phase);
        writeReference(ra, b, phase);
        return b;
    [AssignmentExpression□0 □ PostfixExpression CompoundAssignment AssignmentExpression□1] do
        if phase = compile then throw compileExpressionError end if;
        rLeft: OBJORREF □ Eval[PostfixExpression](env, phase);
        oLeft: OBJECT □ readReference(rLeft, phase);
        rRight: OBJORREF □ Eval[AssignmentExpression□1](env, phase);
        oRight: OBJECT □ readReference(rRight, phase);
        result: OBJECT □ Op[CompoundAssignment](oLeft, oRight, phase);
        writeReference(rLeft, result, phase);
        return result;
    [AssignmentExpression□0 □ PostfixExpression LogicalAssignment AssignmentExpression□1] do
        if phase = compile then throw compileExpressionError end if;
        rLeft: OBJORREF □ Eval[PostfixExpression](env, phase);
        oLeft: OBJECT □ readReference(rLeft, phase);
        bLeft: BOOLEAN □ toBoolean(oLeft, phase);
        result: OBJECT □ oLeft;
        case Operator[LogicalAssignment] of
            {andEq} do
                if bLeft then
                    result □ readReference(Eval[AssignmentExpression□1](env, phase), phase)
                end if;
            {xorEq} do
                bRight: BOOLEAN □ toBoolean(readReference(Eval[AssignmentExpression□1](env, phase), phase), phase);
                result □ bLeft xor bRight;
            {orEq} do
                if not bLeft then
                    result □ readReference(Eval[AssignmentExpression□1](env, phase), phase)
                end if
        end case;
        writeReference(rLeft, result, phase);
        return result
end proc;
```

Op[*CompoundAssignment*]: OBJECT □ OBJECT □ PHASE □ OBJECT;
Op[*CompoundAssignment* □ `*=`] = *multiply*;
Op[*CompoundAssignment* □ `/=`] = *divide*;
Op[*CompoundAssignment* □ `%=`] = *remainder*;
Op[*CompoundAssignment* □ `+=`] = *add*;
Op[*CompoundAssignment* □ `-=`] = *subtract*;
Op[*CompoundAssignment* □ `<<=`] = *shiftLeft*;
Op[*CompoundAssignment* □ `>>=`] = *shiftRight*;
Op[*CompoundAssignment* □ `>>>=`] = *shiftRightUnsigned*;
Op[*CompoundAssignment* □ `&=`] = *bitAnd*;
Op[*CompoundAssignment* □ `^=`] = *bitXor*;
Op[*CompoundAssignment* □ `|=`] = *bitOr*;

Operator[*LogicalAssignment*]: {**andEq**, **xorEq**, **orEq**};
Operator[*LogicalAssignment* □ `&&=`] = **andEq**;
Operator[*LogicalAssignment* □ `^^=`] = **xorEq**;
Operator[*LogicalAssignment* □ `||=`] = **orEq**;

12.20 Comma Expressions

Syntax

ListExpression□ □
AssignmentExpression□
| *ListExpression*□ `,` *AssignmentExpression*□

OptionalExpression □
ListExpression^{allowIn}
| «empty»

Validation

proc Validate[*ListExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ListExpression*□ □ *AssignmentExpression*□] **do**
Validate[*AssignmentExpression*□](*cxt*, *env*);
[*ListExpression*□$_0$ □ *ListExpression*□$_1$ `,` *AssignmentExpression*□] **do**
Validate[*ListExpression*□$_1$](*cxt*, *env*);
Validate[*AssignmentExpression*□](*cxt*, *env*)
end proc;

Evaluation

proc Eval[*ListExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*ListExpression*□ □ *AssignmentExpression*□] **do**
return Eval[*AssignmentExpression*□](*env*, *phase*);
[*ListExpression*□$_0$ □ *ListExpression*□$_1$ `,` *AssignmentExpression*□] **do**
ra: OBJORREF □ Eval[*ListExpression*□$_1$](*env*, *phase*);
readReference(*ra*, *phase*);
rb: OBJORREF □ Eval[*AssignmentExpression*□](*env*, *phase*);
return *readReference*(*rb*, *phase*)
end proc;

```
proc EvalAsList[ListExpression□] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
    [ListExpression□ □ AssignmentExpression□] do
        r: OBJORREF □ Eval[AssignmentExpression□](env, phase);
        elt: OBJECT □ readReference(r, phase);
        return [elt];
    [ListExpression□0 □ ListExpression□1 , AssignmentExpression□] do
        elts: OBJECT[] □ EvalAsList[ListExpression□1](env, phase);
        r: OBJORREF □ Eval[AssignmentExpression□](env, phase);
        elt: OBJECT □ readReference(r, phase);
        return elts ⊕ [elt]
end proc;
```

12.21 Type Expressions

Syntax

```
TypeExpression□ □ NonAssignmentExpression□
```

Validation

```
proc Validate[TypeExpression□ □ NonAssignmentExpression□] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[NonAssignmentExpression□](cxt, env)
end proc;
```

Evaluation

```
proc Eval[TypeExpression□ □ NonAssignmentExpression□] (env: ENVIRONMENT): CLASS
    r: OBJORREF □ Eval[NonAssignmentExpression□](env, compile);
    o: OBJECT □ readReference(r, compile);
    if o □ CLASS then throw badValueError end if;
    return o
end proc;
```

13 Statements

Syntax

```
□ □ {abbrev, noShortIf, full}
```

Statement□ □
  ExpressionStatement Semicolon□
| *SuperStatement Semicolon*□
| *Block*
| *LabeledStatement*□
| *IfStatement*□
| *SwitchStatement*
| *DoStatement Semicolon*□
| *WhileStatement*□
| *ForStatement*□
| *WithStatement*□
| *ContinueStatement Semicolon*□
| *BreakStatement Semicolon*□
| *ReturnStatement Semicolon*□
| *ThrowStatement Semicolon*□
| *TryStatement*

Substatement□ □
  EmptyStatement
| *Statement*□
| *SimpleVariableDefinition Semicolon*□
| *Attributes* [no line break] **{** *Substatements* **}**

Substatements □
  «empty»
| *SubstatementsPrefix Substatement*^{abbrev}

SubstatementsPrefix □
  «empty»
| *SubstatementsPrefix Substatement*^{full}

Semicolon^{abbrev} □
  ;
| **VirtualSemicolon**
| «empty»

Semicolon^{noShortIf} □
  ;
| **VirtualSemicolon**
| «empty»

Semicolon^{full} □
  ;
| **VirtualSemicolon**

Validation

proc Validate[*Statement*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *sl*: LABEL{}, *jt*: JUMPTARGETS, *pl*: PLURALITY)
  [*Statement*□ □ *ExpressionStatement Semicolon*□] **do**
    Validate[*ExpressionStatement*](*cxt*, *env*);

```
    [Statement□ ⇒ SuperStatement Semicolon□] do Validate[SuperStatement](cxt, env);
    [Statement□ ⇒ Block] do Validate[Block](cxt, env, jt, pl);
    [Statement□ ⇒ LabeledStatement□] do Validate[LabeledStatement□](cxt, env, sl, jt);
    [Statement□ ⇒ IfStatement□] do Validate[IfStatement□](cxt, env, jt);
    [Statement□ ⇒ SwitchStatement] do ????;
    [Statement□ ⇒ DoStatement Semicolon□] do Validate[DoStatement](cxt, env, sl, jt);
    [Statement□ ⇒ WhileStatement□] do Validate[WhileStatement□](cxt, env, sl, jt);
    [Statement□ ⇒ ForStatement□] do ????;
    [Statement□ ⇒ WithStatement□] do ????;
    [Statement□ ⇒ ContinueStatement Semicolon□] do Validate[ContinueStatement](jt);
    [Statement□ ⇒ BreakStatement Semicolon□] do Validate[BreakStatement](jt);
    [Statement□ ⇒ ReturnStatement Semicolon□] do Validate[ReturnStatement](cxt, env);
    [Statement□ ⇒ ThrowStatement Semicolon□] do Validate[ThrowStatement](cxt, env);
    [Statement□ ⇒ TryStatement] do ????
end proc;

Enabled[Substatement□]: BOOLEAN;

proc Validate[Substatement□] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
    [Substatement□ ⇒ EmptyStatement] do nothing;
    [Substatement□ ⇒ Statement□] do Validate[Statement□](cxt, env, sl, jt, plural);
    [Substatement□ ⇒ SimpleVariableDefinition Semicolon□] do
        Validate[SimpleVariableDefinition](cxt, env);
    [Substatement□ ⇒ Attributes [no line break] { Substatements }] do
        Validate[Attributes](cxt, env);
        attr: ATTRIBUTE ← Eval[Attributes](env, compile);
        if attr ∉ BOOLEAN then throw badValueError end if;
        Enabled[Substatement□] ← attr;
        if attr then Validate[Substatements](cxt, env, jt) end if
end proc;

proc Validate[Substatements] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [Substatements ⇒ «empty»] do nothing;
    [Substatements ⇒ SubstatementsPrefix Substatement^abbrev] do
        Validate[SubstatementsPrefix](cxt, env, jt);
        Validate[Substatement^abbrev](cxt, env, {}, jt)
end proc;

proc Validate[SubstatementsPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [SubstatementsPrefix ⇒ «empty»] do nothing;
    [SubstatementsPrefix_0 ⇒ SubstatementsPrefix_1 Substatement^full] do
        Validate[SubstatementsPrefix_1](cxt, env, jt);
        Validate[Substatement^full](cxt, env, {}, jt)
end proc;
```

Evaluation

```
proc Eval[Statement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [Statement□ ⇒ ExpressionStatement Semicolon□] do
        return Eval[ExpressionStatement](env);
```

[*Statement*□ □ *SuperStatement Semicolon*□] **do return** Eval[*SuperStatement*](*env*);
[*Statement*□ □ *Block*] **do return** Eval[*Block*](*env*, *d*);
[*Statement*□ □ *LabeledStatement*□] **do return** Eval[*LabeledStatement*□](*env*, *d*);
[*Statement*□ □ *IfStatement*□] **do return** Eval[*IfStatement*□](*env*, *d*);
[*Statement*□ □ *SwitchStatement*] **do** ????;
[*Statement*□ □ *DoStatement Semicolon*□] **do return** Eval[*DoStatement*](*env*, *d*);
[*Statement*□ □ *WhileStatement*□] **do return** Eval[*WhileStatement*□](*env*, *d*);
[*Statement*□ □ *ForStatement*□] **do** ????;
[*Statement*□ □ *WithStatement*□] **do** ????;
[*Statement*□ □ *ContinueStatement Semicolon*□] **do**
return Eval[*ContinueStatement*](*env*, *d*);
[*Statement*□ □ *BreakStatement Semicolon*□] **do return** Eval[*BreakStatement*](*env*, *d*);
[*Statement*□ □ *ReturnStatement Semicolon*□] **do return** Eval[*ReturnStatement*](*env*);
[*Statement*□ □ *ThrowStatement Semicolon*□] **do return** Eval[*ThrowStatement*](*env*);
[*Statement*□ □ *TryStatement*] **do** ????
end proc;

proc Eval[*Substatement*□] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*Substatement*□ □ *EmptyStatement*] **do return** *d*;
[*Substatement*□ □ *Statement*□] **do return** Eval[*Statement*□](*env*, *d*);
[*Substatement*□ □ *SimpleVariableDefinition Semicolon*□] **do**
return Eval[*SimpleVariableDefinition*](*env*, *d*);
[*Substatement*□ □ *Attributes* [no line break] { *Substatements* }] **do**
if Enabled[*Substatement*□] **then return** Eval[*Substatements*](*env*, *d*)
else return *d*
end if
end proc;

proc Eval[*Substatements*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*Substatements* □ «empty»] **do return** *d*;
[*Substatements* □ *SubstatementsPrefix Substatement*^{abbrev}] **do**
o: OBJECT □ Eval[*SubstatementsPrefix*](*env*, *d*);
return Eval[*Substatement*^{abbrev}](*env*, *o*)
end proc;

proc Eval[*SubstatementsPrefix*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*SubstatementsPrefix* □ «empty»] **do return** *d*;
[*SubstatementsPrefix*$_0$ □ *SubstatementsPrefix*$_1$ *Substatement*^{full}] **do**
o: OBJECT □ Eval[*SubstatementsPrefix*$_1$](*env*, *d*);
return Eval[*Substatement*^{full}](*env*, *o*)
end proc;

13.1 Empty Statement

Syntax

EmptyStatement □ ;

13.2 Expression Statement

Syntax

ExpressionStatement ⇒ [lookahead∉{**function**, **{**}] *ListExpression*^{allowIn}

Validation

```
proc Validate[ExpressionStatement ⇒ [lookahead∉{function, {}] ListExpression^allowIn]
        (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[ListExpression^allowIn](cxt, env)
end proc;
```

Evaluation

```
proc Eval[ExpressionStatement ⇒ [lookahead∉{function, {}] ListExpression^allowIn] (env: ENVIRONMENT): OBJECT
    r: OBJORREF ⇒ Eval[ListExpression^allowIn](env, run);
    return readReference(r, run)
end proc;
```

13.3 Super Statement

Syntax

SuperStatement ⇒ **super** *Arguments*

Validation

```
proc Validate[SuperStatement ⇒ super Arguments] (cxt: CONTEXT, env: ENVIRONMENT)
    ????
end proc;
```

Evaluation

```
proc Eval[SuperStatement ⇒ super Arguments] (env: ENVIRONMENT): OBJECT
    ????
end proc;
```

13.4 Block Statement

Syntax

Block ⇒ **{** *Directives* **}**

Validation

```
proc Validate[Block ⇒ { Directives }] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY)
    compileFrame: BLOCKFRAME ⇒
        new BLOCKFRAME⟦staticReadBindings: {}, staticWriteBindings: {}, plurality: pl⟧
    CompileFrame[Block] ⇒ compileFrame;
    Validate[Directives](cxt, [compileFrame] ⊕ env, jt, pl, none)
end proc;
```

```
proc ValidateUsingFrame[Block ⊆ { Directives }]
        (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY, frame: FRAME)
    Validate[Directives](cxt, [frame] ⊕ env, jt, pl, none)
end proc;
```

Evaluation

```
proc Eval[Block ⊆ { Directives }] (env: ENVIRONMENT, d: OBJECT): OBJECT
    compileFrame: BLOCKFRAME ← CompileFrame[Block];
    runtimeFrame: BLOCKFRAME;
    case compileFrame.plurality of
        {singular} do runtimeFrame ← compileFrame;
        {plural} do
            runtimeFrame ← new BLOCKFRAME[staticReadBindings: {}, staticWriteBindings: {}, plurality: singular];
            instantiateFrame(compileFrame, runtimeFrame, [runtimeFrame] ⊕ env)
    end case;
    return Eval[Directives]([runtimeFrame] ⊕ env, d)
end proc;

proc EvalUsingFrame[Block ⊆ { Directives }] (env: ENVIRONMENT, frame: FRAME, d: OBJECT): OBJECT
    return Eval[Directives]([frame] ⊕ env, d)
end proc;

CompileFrame[Block]: BLOCKFRAME;
```

13.5 Labeled Statements

Syntax

```
LabeledStatement⊆ ⊆ Identifier : Substatement⊆
```

Validation

```
proc Validate[LabeledStatement⊆ ⊆ Identifier : Substatement⊆]
        (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
    name: STRING ← Name[Identifier];
    if name ∈ jt.breakTargets then throw syntaxError end if;
    jt2: JUMPTARGETS ← JUMPTARGETS[breakTargets: jt.breakTargets ∪ {name},
            continueTargets: jt.continueTargets];
    Validate[Substatement⊆](cxt, env, sl ∪ {name}, jt2)
end proc;
```

Evaluation

```
proc Eval[LabeledStatement⊆ ⊆ Identifier : Substatement⊆] (env: ENVIRONMENT, d: OBJECT): OBJECT
    try return Eval[Substatement⊆](env, d)
    catch x: SEMANTICEXCEPTION do
        if x ∈ BREAK and x.label = Name[Identifier] then return x.value
        else throw x
        end if
    end try
end proc;
```

13.6 If Statement

Syntax

IfStatement^{abbrev} ⊐
  if *ParenListExpression Substatement*^{abbrev}
 | **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{abbrev}

IfStatement^{full} ⊐
  if *ParenListExpression Substatement*^{full}
 | **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{full}

IfStatement^{noShortIf} ⊐ **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{noShortIf}

Validation

```
proc Validate[IfStatement⊐] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [IfStatement^abbrev ⊐ if ParenListExpression Substatement^abbrev] do
    Validate[ParenListExpression](cxt, env);
    Validate[Substatement^abbrev](cxt, env, {}, jt);
  [IfStatement^full ⊐ if ParenListExpression Substatement^full] do
    Validate[ParenListExpression](cxt, env);
    Validate[Substatement^full](cxt, env, {}, jt);
  [IfStatement⊐ ⊐ if ParenListExpression Substatement^noShortIf_1 else Substatement⊐_2] do
    Validate[ParenListExpression](cxt, env);
    Validate[Substatement^noShortIf_1](cxt, env, {}, jt);
    Validate[Substatement⊐_2](cxt, env, {}, jt)
end proc;
```

Evaluation

```
proc Eval[IfStatement⊐] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [IfStatement^abbrev ⊐ if ParenListExpression Substatement^abbrev] do
    r: OBJORREF ⊐ Eval[ParenListExpression](env, run);
    o: OBJECT ⊐ readReference(r, run);
    if toBoolean(o, run) then return Eval[Substatement^abbrev](env, d)
    else return d
    end if;
  [IfStatement^full ⊐ if ParenListExpression Substatement^full] do
    r: OBJORREF ⊐ Eval[ParenListExpression](env, run);
    o: OBJECT ⊐ readReference(r, run);
    if toBoolean(o, run) then return Eval[Substatement^full](env, d)
    else return d
    end if;
  [IfStatement⊐ ⊐ if ParenListExpression Substatement^noShortIf_1 else Substatement⊐_2] do
    r: OBJORREF ⊐ Eval[ParenListExpression](env, run);
    o: OBJECT ⊐ readReference(r, run);
    if toBoolean(o, run) then return Eval[Substatement^noShortIf_1](env, d)
    else return Eval[Substatement⊐_2](env, d)
    end if
end proc;
```

13.7 Switch Statement

Syntax

SwitchStatement → **switch** *ParenListExpression* **{** *CaseStatements* **}**

CaseStatements →
  «empty»
| *CaseLabel*
| *CaseLabel CaseStatementsPrefix CaseStatement*[abbrev]

CaseStatementsPrefix →
  «empty»
| *CaseStatementsPrefix CaseStatement*[full]

CaseStatement[] →
  Substatement[]
| *CaseLabel*

CaseLabel →
  case *ListExpression*[allowIn] **:**
| **default :**

13.8 Do-While Statement

Syntax

DoStatement → **do** *Substatement*[abbrev] **while** *ParenListExpression*

Validation

Labels[*DoStatement*]: LABEL{};

proc Validate[*DoStatement* → **do** *Substatement*[abbrev] **while** *ParenListExpression*]
  (*cxt*: CONTEXT, *env*: ENVIRONMENT, *sl*: LABEL{}, *jt*: JUMPTARGETS)
  continueLabels: LABEL{} ← *sl* ∪ {**default**};
  Labels[*DoStatement*] ← *continueLabels*;
  jt2: JUMPTARGETS ← JUMPTARGETS[breakTargets: *jt*.breakTargets ∪ {**default**},
    continueTargets: *jt*.continueTargets ∪ *continueLabels*];
  Validate[*Substatement*[abbrev]](*cxt*, *env*, {}, *jt2*);
  Validate[*ParenListExpression*](*cxt*, *env*)
end proc;

Evaluation

```
proc Eval[DoStatement ≡ do Substatement^abbrev while ParenListExpression]
        (env: ENVIRONMENT, d: OBJECT): OBJECT
    try
        d1: OBJECT ← d;
        while true do
            try d1 ← Eval[Substatement^abbrev](env, d1)
            catch x: SEMANTICEXCEPTION do
                if x ∈ CONTINUE and x.label ∈ Labels[DoStatement] then d1 ← x.value
                else throw x
                end if
            end try;
            r: OBJORREF ← Eval[ParenListExpression](env, run);
            o: OBJECT ← readReference(r, run);
            if not toBoolean(o, run) then return d1 end if
        end while
    catch x: SEMANTICEXCEPTION do
        if x ∈ BREAK and x.label = default then return x.value else throw x end if
    end try
end proc;
```

13.9 While Statement

Syntax

```
WhileStatement^ω ≡ while ParenListExpression Substatement^ω
```

Validation

```
Labels[WhileStatement^ω]: LABEL{};

proc Validate[WhileStatement^ω ≡ while ParenListExpression Substatement^ω]
        (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
    Validate[ParenListExpression](cxt, env);
    continueLabels: LABEL{} ← sl ∪ {default};
    Labels[WhileStatement^ω] ← continueLabels;
    jt2: JUMPTARGETS ← JUMPTARGETS[breakTargets: jt.breakTargets ∪ {default},
            continueTargets: jt.continueTargets ∪ continueLabels];
    Validate[Substatement^ω](cxt, env, {}, jt2)
end proc;
```

Evaluation

```
proc Eval[WhileStatement□ □ while ParenListExpression Substatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
    try
        d1: OBJECT □ d;
        while toBoolean(readReference(Eval[ParenListExpression](env, run), run), run) do
            try d1 □ Eval[Substatement□](env, d1)
            catch x: SEMANTICEXCEPTION do
                if x □ CONTINUE and x.label □ Labels[WhileStatement□] then
                    d1 □ x.value
                else throw x
                end if
            end try
        end while;
        return d1
    catch x: SEMANTICEXCEPTION do
        if x □ BREAK and x.label = default then return x.value else throw x end if
    end try
end proc;
```

13.10 For Statements

Syntax

ForStatement□ □
  for (*ForInitialiser* ; *OptionalExpression* ; *OptionalExpression*) *Substatement□*
| **for** (*ForInBinding* **in** *ListExpression*^{allowIn}) *Substatement□*

ForInitialiser □
  «empty»
| *ListExpression*^{noIn}
| *VariableDefinitionKind VariableBindingList*^{noIn}
| *Attributes* [no line break] *VariableDefinitionKind VariableBindingList*^{noIn}

ForInBinding □
  PostfixExpression
| *VariableDefinitionKind VariableBinding*^{noIn}
| *Attributes* [no line break] *VariableDefinitionKind VariableBinding*^{noIn}

13.11 With Statement

Syntax

WithStatement□ □ **with** *ParenListExpression Substatement□*

13.12 Continue and Break Statements

Syntax

ContinueStatement □
  continue
| **continue** [no line break] *Identifier*

BreakStatement □
  break
| **break** [no line break] *Identifier*

Validation

proc Validate[*ContinueStatement*] (*jt*: JUMPTARGETS)
[*ContinueStatement* ⇒ `continue`] **do**
if default ∈ *jt*.continueTargets **then throw syntaxError end if**;
[*ContinueStatement* ⇒ `continue` [no line break] *Identifier*] **do**
if Name[*Identifier*] ∈ *jt*.continueTargets **then throw syntaxError end if**
end proc;

proc Validate[*BreakStatement*] (*jt*: JUMPTARGETS)
[*BreakStatement* ⇒ `break`] **do**
if default ∈ *jt*.breakTargets **then throw syntaxError end if**;
[*BreakStatement* ⇒ `break` [no line break] *Identifier*] **do**
if Name[*Identifier*] ∈ *jt*.breakTargets **then throw syntaxError end if**
end proc;

Evaluation

proc Eval[*ContinueStatement*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*ContinueStatement* ⇒ `continue`] **do throw** CONTINUE{value: *d*, label: **default**}
[*ContinueStatement* ⇒ `continue` [no line break] *Identifier*] **do**
throw CONTINUE{value: *d*, label: Name[*Identifier*]}
end proc;

proc Eval[*BreakStatement*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*BreakStatement* ⇒ `break`] **do throw** BREAK{value: *d*, label: **default**}
[*BreakStatement* ⇒ `break` [no line break] *Identifier*] **do**
throw BREAK{value: *d*, label: Name[*Identifier*]}
end proc;

13.13 Return Statement

Syntax

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

Validation

proc Validate[*ReturnStatement*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ReturnStatement* ⇒ `return`] **do**
if *getRegionalFrame*(*env*) ∈ PARAMETERFRAME **then throw syntaxError end if**;
[*ReturnStatement* ⇒ `return` [no line break] *ListExpression*^{allowIn}] **do**
if *getRegionalFrame*(*env*) ∈ PARAMETERFRAME **then throw syntaxError end if**;
Validate[*ListExpression*^{allowIn}](*cxt*, *env*)
end proc;

Evaluation

```
proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
    [ReturnStatement ⇒ return] do throw RETURNEDVALUE{value: undefined}
    [ReturnStatement ⇒ return [no line break] ListExpression^allowIn] do
        r: OBJORREF ← Eval[ListExpression^allowIn](env, run);
        a: OBJECT ← readReference(r, run);
        throw RETURNEDVALUE{value: a}
end proc;
```

13.14 Throw Statement

Syntax

ThrowStatement ⇒ **throw** [no line break] *ListExpression*^{allowIn}

Validation

```
Validate[ThrowStatement ⇒ throw [no line break] ListExpression^allowIn]: CONTEXT → ENVIRONMENT → ()
        = Validate[ListExpression^allowIn];
```

Evaluation

```
proc Eval[ThrowStatement ⇒ throw [no line break] ListExpression^allowIn] (env: ENVIRONMENT): OBJECT
    r: OBJORREF ← Eval[ListExpression^allowIn](env, run);
    a: OBJECT ← readReference(r, run);
    throw THROWNVALUE{value: a}
end proc;
```

13.15 Try Statement

Syntax

TryStatement ⇒
  try *Block CatchClauses*
| **try** *Block FinallyClause*
| **try** *Block CatchClauses FinallyClause*

CatchClauses ⇒
  CatchClause
| *CatchClauses CatchClause*

CatchClause ⇒ **catch** (*Parameter*) *Block*

FinallyClause ⇒ **finally** *Block*

14 Directives

Syntax

Directive[] ⊐
  EmptyStatement
| *Statement*[]
| *AnnotatableDirective*[]
| *Attributes* [no line break] *AnnotatableDirective*[]
| *Attributes* [no line break] **{** *Directives* **}**
| *PackageDefinition*
| *Pragma Semicolon*[]

AnnotatableDirective[] ⊐
  ExportDefinition Semicolon[]
| *VariableDefinition Semicolon*[]
| *FunctionDefinition*
| *ClassDefinition*
| *NamespaceDefinition Semicolon*[]
| *ImportDirective Semicolon*[]
| *UseDirective Semicolon*[]

Directives ⊐
  «empty»
| *DirectivesPrefix Directive*[abbrev]

DirectivesPrefix ⊐
  «empty»
| *DirectivesPrefix Directive*[full]

Validation

```
proc Validate[Directive[]] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
      attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
    [Directive[] ⊐ EmptyStatement] do return cxt;
    [Directive[] ⊐ Statement[]] do
      if attr ⊐ {none, true} then throw syntaxError end if;
      Validate[Statement[]](cxt, env, {}, jt, pl);
      return cxt;
    [Directive[] ⊐ AnnotatableDirective[]] do
      return Validate[AnnotatableDirective[]](cxt, env, pl, attr);
    [Directive[] ⊐ Attributes [no line break] AnnotatableDirective[]] do
      Validate[Attributes](cxt, env);
      attr2: ATTRIBUTE ⊐ Eval[Attributes](env, compile);
      attr3: ATTRIBUTE ⊐ combineAttributes(attr, attr2);
      Enabled[Directive[]] ⊐ attr3 ≠ false;
      if attr3 ≠ false then return Validate[AnnotatableDirective[]](cxt, env, pl, attr3)
      else return cxt
      end if;
```

```
    [Directive□ □ Attributes [no line break] { Directives }] do
        Validate[Attributes](cxt, env);
        attr2: ATTRIBUTE □ Eval[Attributes](env, compile);
        attr3: ATTRIBUTE □ combineAttributes(attr, attr2);
        Enabled[Directive□] □ attr3 ≠ false;
        if attr3 = false then return cxt end if;
        return Validate[Directives](cxt, env, jt, pl, attr3);
    [Directive□ □ PackageDefinition] do
        if attr □ {none, true} then ???? else throw syntaxError end if;
    [Directive□ □ Pragma Semicolon□] do
        if attr □ {none, true} then return Validate[Pragma](cxt)
        else throw syntaxError
        end if
end proc;

proc Validate[AnnotatableDirective□]
        (cxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
    [AnnotatableDirective□ □ ExportDefinition Semicolon□] do ????;
    [AnnotatableDirective□ □ VariableDefinition Semicolon□] do
        Validate[VariableDefinition](cxt, env, attr);
        return cxt;
    [AnnotatableDirective□ □ FunctionDefinition] do
        Validate[FunctionDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective□ □ ClassDefinition] do
        Validate[ClassDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective□ □ NamespaceDefinition Semicolon□] do
        Validate[NamespaceDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective□ □ ImportDirective Semicolon□] do ????;
    [AnnotatableDirective□ □ UseDirective Semicolon□] do
        if attr □ {none, true} then return Validate[UseDirective](cxt, env)
        else throw syntaxError
        end if
end proc;

proc Validate[Directives] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
        attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
    [Directives □ «empty»] do return cxt;
    [Directives □ DirectivesPrefix Directive^abbrev] do
        cxt2: CONTEXT □ Validate[DirectivesPrefix](cxt, env, jt, pl, attr);
        return Validate[Directive^abbrev](cxt2, env, jt, pl, attr)
end proc;

proc Validate[DirectivesPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
        attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
    [DirectivesPrefix □ «empty»] do return cxt;
    [DirectivesPrefix_0 □ DirectivesPrefix_1 Directive^full] do
        cxt2: CONTEXT □ Validate[DirectivesPrefix_1](cxt, env, jt, pl, attr);
        return Validate[Directive^full](cxt2, env, jt, pl, attr)
end proc;
```

Evaluation

proc Eval[*Directive*□] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*Directive*□ □ *EmptyStatement*] **do return** *d*;
[*Directive*□ □ *Statement*□] **do return** Eval[*Statement*□](*env*, *d*);
[*Directive*□ □ *AnnotatableDirective*□] **do return** Eval[*AnnotatableDirective*□](*env*, *d*);
[*Directive*□ □ *Attributes* [no line break] *AnnotatableDirective*□] **do**
if Enabled[*Directive*□] **then return** Eval[*AnnotatableDirective*□](*env*, *d*)
else return *d*
end if;
[*Directive*□ □ *Attributes* [no line break] { *Directives* }] **do**
if Enabled[*Directive*□] **then return** Eval[*Directives*](*env*, *d*) **else return** *d* **end if**;
[*Directive*□ □ *PackageDefinition*] **do** ????;
[*Directive*□ □ *Pragma Semicolon*□] **do return** *d*
end proc;

proc Eval[*AnnotatableDirective*□] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*AnnotatableDirective*□ □ *ExportDefinition Semicolon*□] **do** ????;
[*AnnotatableDirective*□ □ *VariableDefinition Semicolon*□] **do**
return Eval[*VariableDefinition*](*env*, *d*);
[*AnnotatableDirective*□ □ *FunctionDefinition*] **do return** *d*;
[*AnnotatableDirective*□ □ *ClassDefinition*] **do return** Eval[*ClassDefinition*](*env*, *d*);
[*AnnotatableDirective*□ □ *NamespaceDefinition Semicolon*□] **do return** *d*;
[*AnnotatableDirective*□ □ *ImportDirective Semicolon*□] **do** ????;
[*AnnotatableDirective*□ □ *UseDirective Semicolon*□] **do return** *d*
end proc;

proc Eval[*Directives*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*Directives* □ «empty»] **do return** *d*;
[*Directives* □ *DirectivesPrefix Directive*^{abbrev}] **do**
o: OBJECT □ Eval[*DirectivesPrefix*](*env*, *d*);
return Eval[*Directive*^{abbrev}](*env*, *o*)
end proc;

proc Eval[*DirectivesPrefix*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
[*DirectivesPrefix* □ «empty»] **do return** *d*;
[*DirectivesPrefix*$_0$ □ *DirectivesPrefix*$_1$ *Directive*^{full}] **do**
o: OBJECT □ Eval[*DirectivesPrefix*$_1$](*env*, *d*);
return Eval[*Directive*^{full}](*env*, *o*)
end proc;

Enabled[*Directive*□]: BOOLEAN;

14.1 Attributes

Syntax

Attributes □
Attribute
| *AttributeCombination*

AttributeCombination □ *Attribute* [no line break] *Attributes*

Attribute ⊐
  AttributeExpression
| **true**
| **false**
| **public**
| *NonexpressionAttribute*

NonexpressionAttribute ⊐
  final
| **private**
| **static**

Validation

proc Validate[*Attributes*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*Attributes* ⊐ *Attribute*] **do** Validate[*Attribute*](*cxt*, *env*);
  [*Attributes* ⊐ *AttributeCombination*] **do** Validate[*AttributeCombination*](*cxt*, *env*)
end proc;

proc Validate[*AttributeCombination* ⊐ *Attribute* [no line break] *Attributes*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*Attribute*](*cxt*, *env*);
  Validate[*Attributes*](*cxt*, *env*)
end proc;

proc Validate[*Attribute*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*Attribute* ⊐ *AttributeExpression*] **do** Validate[*AttributeExpression*](*cxt*, *env*);
  [*Attribute* ⊐ **true**] **do nothing**;
  [*Attribute* ⊐ **false**] **do nothing**;
  [*Attribute* ⊐ **public**] **do nothing**;
  [*Attribute* ⊐ *NonexpressionAttribute*] **do** Validate[*NonexpressionAttribute*](*env*)
end proc;

proc Validate[*NonexpressionAttribute*] (*env*: ENVIRONMENT)
  [*NonexpressionAttribute* ⊐ **final**] **do nothing**;
  [*NonexpressionAttribute* ⊐ **private**] **do**
    if *getEnclosingClass*(*env*) = **none then throw syntaxError end if**;
  [*NonexpressionAttribute* ⊐ **static**] **do nothing**
end proc;

Evaluation

proc Eval[*Attributes*] (*env*: ENVIRONMENT, *phase*: PHASE): ATTRIBUTE
  [*Attributes* ⊐ *Attribute*] **do return** Eval[*Attribute*](*env*, *phase*);
  [*Attributes* ⊐ *AttributeCombination*] **do return** Eval[*AttributeCombination*](*env*, *phase*)
end proc;

proc Eval[*AttributeCombination* ⊐ *Attribute* [no line break] *Attributes*]
    (*env*: ENVIRONMENT, *phase*: PHASE): ATTRIBUTE
  a: ATTRIBUTE ⊐ Eval[*Attribute*](*env*, *phase*);
  if *a* = **false then return false end if**;
  b: ATTRIBUTE ⊐ Eval[*Attributes*](*env*, *phase*);
  return *combineAttributes*(*a*, *b*)
end proc;

```
proc Eval[Attribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
    [Attribute ⊆ AttributeExpression] do
        r: OBJORREF ← Eval[AttributeExpression](env, phase);
        a: OBJECT ← readReference(r, phase);
        if a ∉ ATTRIBUTE then throw badValueError end if;
        return a;
    [Attribute ⊆ true] do return true;
    [Attribute ⊆ false] do return false;
    [Attribute ⊆ public] do return publicNamespace;
    [Attribute ⊆ NonexpressionAttribute] do
        return Eval[NonexpressionAttribute](env, phase)
end proc;

proc Eval[NonexpressionAttribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
    [NonexpressionAttribute ⊆ final] do
        return COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: false, dynamic: false, memberMod: final,
                overrideMod: none, prototype: false, unused: false⟧
    [NonexpressionAttribute ⊆ private] do
        c: CLASSOPT ← getEnclosingClass(env);
        Note that Validate ensured that c cannot be none at this point.
        return c.privateNamespace;
    [NonexpressionAttribute ⊆ static] do
        return COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: false, dynamic: false, memberMod: static,
                overrideMod: none, prototype: false, unused: false⟧
end proc;
```

14.2 Use Directive

Syntax

```
UseDirective ⊆ use namespace ParenListExpression
```

Validation

```
proc Validate[UseDirective ⊆ use namespace ParenListExpression] (cxt: CONTEXT, env: ENVIRONMENT): CONTEXT
    Validate[ParenListExpression](cxt, env);
    values: OBJECT[] ← EvalAsList[ParenListExpression](env, compile);
    namespaces: NAMESPACE{} ← {};
    for each v ∈ values do
        if v ∉ NAMESPACE or v ∈ namespaces then throw badValueError end if;
        namespaces ← namespaces ∪ {v}
    end for each;
    return CONTEXT⟦openNamespaces: cxt.openNamespaces ∪ namespaces, other fields from cxt⟧
end proc;
```

14.3 Import Directive

Syntax

```
ImportDirective ⊆
    import ImportBinding IncludesExcludes
  | import ImportBinding , namespace ParenListExpression IncludesExcludes
```

ImportBinding ⊐
  ImportSource
| *Identifier* **=** *ImportSource*

ImportSource ⊐
  String
| *PackageName*

IncludesExcludes ⊐
  «empty»
| **, exclude (** *NamePatterns* **)**
| **, include (** *NamePatterns* **)**

NamePatterns ⊐
  «empty»
| *NamePatternList*

NamePatternList ⊐
  QualifiedIdentifier
| *NamePatternList* **,** *QualifiedIdentifier*

14.4 Pragma

Syntax

Pragma ⊐ **use** *PragmaItems*

PragmaItems ⊐
  PragmaItem
| *PragmaItems* **,** *PragmaItem*

PragmaItem ⊐
  PragmaExpr
| *PragmaExpr* **?**

PragmaExpr ⊐
  Identifier
| *Identifier* **(** *PragmaArgument* **)**

PragmaArgument ⊐
  true
| **false**
| **Number**
| **- Number**
| **String**

Validation

proc Validate[*Pragma* ⊐ **use** *PragmaItems*] (*cxt*: CONTEXT): CONTEXT
  return Validate[*PragmaItems*](*cxt*)
end proc;

proc Validate[*PragmaItems*] (*cxt*: CONTEXT): CONTEXT
  [*PragmaItems* ⊐ *PragmaItem*] **do return** Validate[*PragmaItem*](*cxt*);
  [*PragmaItems*$_0$ ⊐ *PragmaItems*$_1$ **,** *PragmaItem*] **do**
    cxt2: CONTEXT ⊐ Validate[*PragmaItems*$_1$](*cxt*);
    return Validate[*PragmaItem*](*cxt2*)
end proc;

```
proc Validate[PragmaItem] (cxt: CONTEXT): CONTEXT
    [PragmaItem ⊆ PragmaExpr] do return Validate[PragmaExpr](cxt, false);
    [PragmaItem ⊆ PragmaExpr ?] do return Validate[PragmaExpr](cxt, true)
end proc;

proc Validate[PragmaExpr] (cxt: CONTEXT, optional: BOOLEAN): CONTEXT
    [PragmaExpr ⊆ Identifier] do
        return processPragma(cxt, Name[Identifier], undefined, optional);
    [PragmaExpr ⊆ Identifier ( PragmaArgument )] do
        arg: OBJECT ⊆ Value[PragmaArgument];
        return processPragma(cxt, Name[Identifier], arg, optional)
end proc;

Value[PragmaArgument]: OBJECT;
    Value[PragmaArgument ⊆ true] = true;
    Value[PragmaArgument ⊆ false] = false;
    Value[PragmaArgument ⊆ Number] = Value[Number];
    Value[PragmaArgument ⊆ - Number] = float64Negate(Value[Number]);
    Value[PragmaArgument ⊆ String] = Value[String];

proc processPragma(cxt: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN): CONTEXT
    if name = "strict" then
        if value ∈ {true, undefined} then
            return CONTEXT[strict: true, other fields from cxt]
        end if;
        if value = false then return CONTEXT[strict: false, other fields from cxt] end if
    end if;
    if name = "ecmascript" then
        if value ∈ {undefined, 4.0} then return cxt end if;
        if value ∈ {1.0, 2.0, 3.0} then
            An implementation may optionally modify cxt to disable features not available in ECMAScript Edition value
                other than subsequent pragmas.
            return cxt
        end if
    end if;
    if optional then return cxt else throw badValueError end if
end proc;
```

15 Definitions

15.1 Export Definition

Syntax

```
ExportDefinition ⊆ export ExportBindingList

ExportBindingList ⊆
    ExportBinding
  | ExportBindingList , ExportBinding

ExportBinding ⊆
    FunctionName
  | FunctionName = FunctionName
```

15.2 Variable Definition

Syntax

VariableDefinition ⇒ *VariableDefinitionKind VariableBindingList*[allowIn]

VariableDefinitionKind ⇒
  var
| **const**

VariableBindingList[] ⇒
  VariableBinding[]
| *VariableBindingList*[] **,** *VariableBinding*[]

Semantics

tag hoisted;

tag instance;

Syntax

VariableBinding[] ⇒ *TypedIdentifier*[] *VariableInitialisation*[]

VariableInitialisation[] ⇒
  «empty»
| **=** *VariableInitialiser*[]

VariableInitialiser[] ⇒
  AssignmentExpression[]
| *NonexpressionAttribute*
| *AttributeCombination*

TypedIdentifier[] ⇒
  Identifier
| *Identifier* **:** *TypeExpression*[]

Validation

proc Validate[*VariableDefinition* ⇒ *VariableDefinitionKind VariableBindingList*[allowIn]]
  (*cxt*: CONTEXT, *env*: ENVIRONMENT, *attr*: ATTRIBUTEOPTNOTFALSE)
  immutable: BOOLEAN ← Immutable[*VariableDefinitionKind*];
  Validate[*VariableBindingList*[allowIn]](*cxt*, *env*, *attr*, *immutable*)
end proc;

Immutable[*VariableDefinitionKind*]: BOOLEAN;
  Immutable[*VariableDefinitionKind* ⇒ **var**] = **false**;
  Immutable[*VariableDefinitionKind* ⇒ **const**] = **true**;

proc Validate[*VariableBindingList*[]]
  (*cxt*: CONTEXT, *env*: ENVIRONMENT, *attr*: ATTRIBUTEOPTNOTFALSE, *immutable*: BOOLEAN)
  [*VariableBindingList*[] ⇒ *VariableBinding*[]] **do**
    Validate[*VariableBinding*[]](*cxt*, *env*, *attr*, *immutable*);
  [*VariableBindingList*[]$_0$ ⇒ *VariableBindingList*[]$_1$ **,** *VariableBinding*[]] **do**
    Validate[*VariableBindingList*[]$_1$](*cxt*, *env*, *attr*, *immutable*);
    Validate[*VariableBinding*[]](*cxt*, *env*, *attr*, *immutable*)
end proc;

Kind[*VariableBinding*□]: {**hoisted**, **static**, **instance**};

Multiname[*VariableBinding*□]: MULTINAME;

```
proc Validate[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□]
      (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN)
   Validate[TypedIdentifier□](cxt, env);
   Validate[VariableInitialisation□](cxt, env);
   name: STRING □ Name[TypedIdentifier□];
   if not cxt.strict and getRegionalFrame(env) □ GLOBAL □ PARAMETERFRAME and not immutable and
         attr = none and not TypePresent[TypedIdentifier□] then
      Kind[VariableBinding□] □ hoisted;
      qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: publicNamespace, id: name□;
      Multiname[VariableBinding□] □ {qname};
      defineHoistedVar(env, name)
   else
      a: COMPOUNDATTRIBUTE □ toCompoundAttribute(attr);
      if a.dynamic or a.prototype then throw definitionError end if;
      memberMod: MEMBERMODIFIER □ a.memberMod;
      if env[0] □ CLASS then if memberMod = none then memberMod □ final end if
      else if memberMod ≠ none then throw definitionError end if
      end if;
      v: VARIABLE □ INSTANCEVARIABLE;
      overriddenRead: OVERRIDDENMEMBER □ none;
      overriddenWrite: OVERRIDDENMEMBER □ none;
      case memberMod of
         {none, static} do
            proc evalType(): CLASS
               type: CLASSOPT □ Eval[TypedIdentifier□](env);
               if type = none then return objectClass end if;
               return type
            end proc;
            proc evalInitialiser(): OBJECT
               value: OBJECTOPT □ Eval[VariableInitialisation□](env, compile);
               if value = none then throw compileExpressionError end if;
               return value
            end proc;
            initialValue: VARIABLEVALUE □ inaccessible;
            if immutable then initialValue □ evalInitialiser end if;
            v □ new VARIABLE□□type: evalType, value: initialValue, immutable: immutable□□;
            multiname: MULTINAME □ defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit,
                  readWrite, v);
            Multiname[VariableBinding□] □ multiname;
            Kind[VariableBinding□] □ static;
         {virtual, final} do
            c: CLASS □ env[0];
            proc evalInitialValue(): OBJECTOPT
               return Eval[VariableInitialisation□](env, run)
            end proc;
            v □ new INSTANCEVARIABLE□□evalInitialValue: evalInitialValue, immutable: immutable,
                  final: memberMod = final□□;
            os: OVERRIDESTATUSPAIR □ defineInstanceMember(c, cxt, name, a.namespaces, a.overrideMod,
                  a.explicit, readWrite, v);
            overriddenRead □ os.readStatus.overriddenMember;
            overriddenWrite □ os.writeStatus.overriddenMember;
            Kind[VariableBinding□] □ instance;
         {constructor} do throw definitionError
      end case;
```

The following sets up PreEval to be called during the pre-evaluation pass.

```
        proc preEvaluate()
            PreEval[VariableBinding□](env, v, overriddenRead, overriddenWrite)
        end proc;
        preEvaluators □ preEvaluators ⊕ [preEvaluate]
    end if
end proc;

proc Validate[VariableInitialisation□] (cxt: CONTEXT, env: ENVIRONMENT)
    [VariableInitialisation□ □ «empty»] do nothing;
    [VariableInitialisation□ □ = VariableInitialiser□] do
        Validate[VariableInitialiser□](cxt, env)
end proc;

proc Validate[VariableInitialiser□] (cxt: CONTEXT, env: ENVIRONMENT)
    [VariableInitialiser□ □ AssignmentExpression□] do
        Validate[AssignmentExpression□](cxt, env);
    [VariableInitialiser□ □ NonexpressionAttribute] do
        Validate[NonexpressionAttribute](env);
    [VariableInitialiser□ □ AttributeCombination] do
        Validate[AttributeCombination](cxt, env)
end proc;

Name[TypedIdentifier□]: STRING;
    Name[TypedIdentifier□ □ Identifier] = Name[Identifier];
    Name[TypedIdentifier□ □ Identifier : TypeExpression□] = Name[Identifier];

TypePresent[TypedIdentifier□]: BOOLEAN;
    TypePresent[TypedIdentifier□ □ Identifier] = false;
    TypePresent[TypedIdentifier□ □ Identifier : TypeExpression□] = true;

proc Validate[TypedIdentifier□] (cxt: CONTEXT, env: ENVIRONMENT)
    [TypedIdentifier□ □ Identifier] do nothing;
    [TypedIdentifier□ □ Identifier : TypeExpression□] do
        Validate[TypeExpression□](cxt, env)
end proc;
```

Pre-Evaluation

```
proc PreEval[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□] (env: ENVIRONMENT,
        v: VARIABLE □ INSTANCEVARIABLE, overriddenRead: OVERRIDDENMEMBER,
        overriddenWrite: OVERRIDDENMEMBER)
    case v of
        VARIABLE do
            type: CLASS □ getVariableType(v, compile);
            value: VARIABLEVALUE □ v.value;
            if value □ () □ OBJECT then
                v.value □ inaccessible;
                try
                    newValue: OBJECT □ value();
                    coercedValue: OBJECT □ assignmentConversion(newValue, type);
                    v.value □ coercedValue
                catch x: SEMANTICEXCEPTION do
                    if x ≠ compileExpressionError then throw x end if;
                    If a compileExpressionError occurred, then the initialiser is not a compile-time constant expression. In
                        this case, ignore the error and leave the value of the variable inaccessible until it is defined at run
                        time.
                end try
            end if;
        INSTANCEVARIABLE do
            t: CLASSOPT □ Eval[TypedIdentifier□](env);
            if t = none then
                if overriddenRead □ {none, potentialConflict} then
                    Note that defineInstanceMember already ensured that overriddenRead □ INSTANCEMETHOD.
                    t □ overriddenRead.type
                elsif overriddenWrite □ {none, potentialConflict} then
                    Note that defineInstanceMember already ensured that overriddenWrite □ INSTANCEMETHOD.
                    t □ overriddenWrite.type
                else t □ objectClass
                end if
            end if;
            v.type □ t
    end case
end proc;
```

Evaluation

```
proc Eval[VariableDefinition □ VariableDefinitionKind VariableBindingList^allowIn]
        (env: ENVIRONMENT, d: OBJECT): OBJECT
    immutable: BOOLEAN □ Immutable[VariableDefinitionKind];
    Eval[VariableBindingList^allowIn](env, immutable);
    return d
end proc;

proc Eval[VariableBindingList□] (env: ENVIRONMENT, immutable: BOOLEAN)
    [VariableBindingList□ □ VariableBinding□] do Eval[VariableBinding□](env, immutable);
    [VariableBindingList□_0 □ VariableBindingList□_1 , VariableBinding□] do
        Eval[VariableBindingList□_1](env, immutable);
        Eval[VariableBinding□](env, immutable)
end proc;
```

```
proc Eval[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□] (env: ENVIRONMENT, immutable: BOOLEAN)
    case Kind[VariableBinding□] of
        {hoisted} do
            value: OBJECTOPT □ Eval[VariableInitialisation□](env, run);
            if value ≠ none then
                lexicalWrite(env, Multiname[VariableBinding□], value, false, run)
            end if;
        {static} do
            localFrame: FRAME □ env[0];
            members: STATICMEMBER{} □ {b.content | □b □ localFrame.staticWriteBindings such that
                    b.qname □ Multiname[VariableBinding□]};
            Note that the members set consists of exactly one VARIABLE element because localFrame was constructed with
                    that VARIABLE inside Validate.
            v: VARIABLE □ the one element of members;
            if v.value = inaccessible then
                value: OBJECTOPT □ Eval[VariableInitialisation□](env, run);
                type: CLASS □ getVariableType(v, run);
                coercedValue: OBJECTU;
                if value ≠ none then coercedValue □ assignmentConversion(value, type)
                elsif immutable then coercedValue □ uninitialised
                else coercedValue □ assignmentConversion(undefined, type)
                end if;
                v.value □ coercedValue
            end if;
        {instance} do nothing
    end case
end proc;

proc Eval[VariableInitialisation□] (env: ENVIRONMENT, phase: PHASE): OBJECTOPT
    [VariableInitialisation□ □ «empty»] do return none;
    [VariableInitialisation□ □ = VariableInitialiser□] do
        return Eval[VariableInitialiser□](env, phase)
end proc;

proc Eval[VariableInitialiser□] (env: ENVIRONMENT, phase: PHASE): OBJECT
    [VariableInitialiser□ □ AssignmentExpression□] do
        r: OBJOREF □ Eval[AssignmentExpression□](env, phase);
        return readReference(r, phase);
    [VariableInitialiser□ □ NonexpressionAttribute] do
        return Eval[NonexpressionAttribute](env, phase);
    [VariableInitialiser□ □ AttributeCombination] do
        return Eval[AttributeCombination](env, phase)
end proc;

proc Eval[TypedIdentifier□] (env: ENVIRONMENT): CLASSOPT
    [TypedIdentifier□ □ Identifier] do return none;
    [TypedIdentifier□ □ Identifier : TypeExpression□] do
        return Eval[TypeExpression□](env)
end proc;
```

15.3 Simple Variable Definition

Syntax

A *SimpleVariableDefinition* represents the subset of *VariableDefinition* expansions that may be used when the variable definition is used as a *Substatement* instead of a *Directive* in non-strict mode. In strict mode variable definitions may not be used as substatements.

SimpleVariableDefinition ⇒ **var** *UntypedVariableBindingList*

UntypedVariableBindingList ⇒
  UntypedVariableBinding
| *UntypedVariableBindingList* **,** *UntypedVariableBinding*

UntypedVariableBinding ⇒ *Identifier VariableInitialisation*^{allowIn}

Validation

```
proc Validate[SimpleVariableDefinition ⇒ var UntypedVariableBindingList] (cxt: CONTEXT, env: ENVIRONMENT)
    if cxt.strict or getRegionalFrame(env) ∈ GLOBAL ∪ PARAMETERFRAME then
        throw syntaxError
    end if;
    Validate[UntypedVariableBindingList](cxt, env)
end proc;

proc Validate[UntypedVariableBindingList] (cxt: CONTEXT, env: ENVIRONMENT)
    [UntypedVariableBindingList ⇒ UntypedVariableBinding] do
        Validate[UntypedVariableBinding](cxt, env);
    [UntypedVariableBindingList0 ⇒ UntypedVariableBindingList1 , UntypedVariableBinding] do
        Validate[UntypedVariableBindingList1](cxt, env);
        Validate[UntypedVariableBinding](cxt, env)
end proc;

proc Validate[UntypedVariableBinding ⇒ Identifier VariableInitialisation^allowIn] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[VariableInitialisation^allowIn](cxt, env);
    defineHoistedVar(env, Name[Identifier])
end proc;
```

Evaluation

```
proc Eval[SimpleVariableDefinition ⇒ var UntypedVariableBindingList] (env: ENVIRONMENT, d: OBJECT): OBJECT
    Eval[UntypedVariableBindingList](env);
    return d
end proc;

proc Eval[UntypedVariableBindingList] (env: ENVIRONMENT)
    [UntypedVariableBindingList ⇒ UntypedVariableBinding] do
        Eval[UntypedVariableBinding](env);
    [UntypedVariableBindingList0 ⇒ UntypedVariableBindingList1 , UntypedVariableBinding] do
        Eval[UntypedVariableBindingList1](env);
        Eval[UntypedVariableBinding](env)
end proc;
```

proc Eval[*UntypedVariableBinding* ⇒ *Identifier VariableInitialisation*^{allowIn}] (*env*: ENVIRONMENT)
  value: OBJECTOPT ← Eval[*VariableInitialisation*^{allowIn}](*env*, **run**);
  if *value* ≠ **none then**
    qname: QUALIFIEDNAME ← QUALIFIEDNAME[namespace: *publicNamespace*, id: Name[*Identifier*]]
    lexicalWrite(*env*, {*qname*}, *value*, **false**, **run**)
  end if
end proc;

15.4 Function Definition

Syntax

FunctionDefinition ⇒ **function** *FunctionName FunctionSignature Block*

FunctionName ⇒
  Identifier
  | **get** [no line break] *Identifier*
  | **set** [no line break] *Identifier*

Validation

```
proc Validate[FunctionDefinition ≡ function FunctionName FunctionSignature Block]
      (cxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE)
   name: STRING ← Name[FunctionName];
   kind: FUNCTIONKIND ← Kind[FunctionName];
   a: COMPOUNDATTRIBUTE ← toCompoundAttribute(attr);
   if a.dynamic then throw definitionError end if;
   unchecked: BOOLEAN ← not cxt.strict and env[0] ∈ CLASS and kind = normal and Untyped[FunctionSignature];
   prototype: BOOLEAN ← unchecked or a.prototype;
   memberMod: MEMBERMODIFIER ← a.memberMod;
   if env[0] ∈ CLASS then if memberMod = none then memberMod ← virtual end if
   else if memberMod ≠ none then throw definitionError end if
   end if;
   if prototype and (kind ≠ normal or memberMod = constructor) then
      throw definitionError
   end if;
   compileThis: {none, inaccessible} ← none;
   if prototype or memberMod ∈ {constructor, virtual, final} then
      compileThis ← inaccessible
   end if;
   compileFrame: PARAMETERFRAME ← new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {},
         plurality: plural, this: compileThis, prototype: prototype];
   compileEnv: ENVIRONMENT ← [compileFrame] ⊕ env;
   Validate[FunctionSignature](cxt, compileEnv);
   Validate[Block](cxt, compileEnv, JUMPTARGETS[breakTargets: {}, continueTargets: {}], plural);
   if unchecked and env[0] ∈ GLOBAL ∈ PARAMETERFRAME and attr = none then
      v: HOISTEDVAR ← new HOISTEDVAR[value: undefined, hasFunctionInitialiser: true];
      defineHoistedVar(env, name);
      ????
   else
      case memberMod of
         {none, static} do
            proc call(this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
               if phase = compile then throw compileExpressionError end if;
               runtimeThis: OBJECTOPT;
               case compileThis of
                  {none} do runtimeThis ← none;
                  {inaccessible} do
                     runtimeThis ← this;
                     g: PACKAGE ∈ GLOBAL ← getPackageOrGlobalFrame(runtimeEnv);
                     if prototype and runtimeThis ∈ {null, undefined} and g ∈ GLOBAL then
                        runtimeThis ← g
                     end if
               end case;
               runtimeFrame: PARAMETERFRAME ← new PARAMETERFRAME[staticReadBindings: {},
                     staticWriteBindings: {}, plurality: singular, this: runtimeThis, prototype: prototype,
                     signature: compileFrame.signature];
               instantiateFrame(compileFrame, runtimeFrame, [runtimeFrame] ⊕ runtimeEnv);
               assignArguments(runtimeFrame, compileFrame.signature, unchecked, args);
               try
                  Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined);
                  throw RETURNEDVALUE[value: undefined]
               catch x: SEMANTICEXCEPTION do
                  if x ∈ RETURNEDVALUE then return x.value else throw x end if
               end try
```

```
            end proc;
            proc construct(args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
                ????
            end proc;
            f: INSTANCE ← OPENINSTANCE;
            if kind ∈ {get, set} then ????
            elsif prototype then ????
            else
                proc instantiate(runtimeEnv: ENVIRONMENT): NONALIASINSTANCE
                    return new FIXEDINSTANCE[type: functionClass, call: call, construct: badConstruct, env: env,
                            typeofString: "Function", slots: {}]
                end proc;
                f ← new OPENINSTANCE[instantiate: instantiate, cache: none]
            end if;
            if pl = singular then f ← instantiateOpenInstance(f, env) end if;
            v: VARIABLE ← new VARIABLE[type: functionClass, value: f, immutable: true];
            defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v);
        {virtual, final} do ????;
        {constructor} do ????
    end case
end if;
The following sets up PreEval to be called during the pre-evaluation pass.
proc preEvaluate()
    PreEval[FunctionDefinition](cxt, compileEnv, compileFrame, unchecked)
end proc;
preEvaluators ← preEvaluators ⊕ [preEvaluate]
end proc;

Kind[FunctionName]: FUNCTIONKIND;
    Kind[FunctionName → Identifier] = normal;
    Kind[FunctionName → get [no line break] Identifier] = get;
    Kind[FunctionName → set [no line break] Identifier] = set;

Name[FunctionName]: STRING;
    Name[FunctionName → Identifier] = Name[Identifier];
    Name[FunctionName → get [no line break] Identifier] = Name[Identifier];
    Name[FunctionName → set [no line break] Identifier] = Name[Identifier];
```

Pre-Evaluation

```
proc PreEval[FunctionDefinition → function FunctionName FunctionSignature Block]
        (cxt: CONTEXT, compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, unchecked: BOOLEAN)
    compileFrame.signature ← PreEval[FunctionSignature](cxt, compileEnv, unchecked)
end proc;

proc assignArguments(runtimeFrame: PARAMETERFRAME, sig: SIGNATURE, unchecked: BOOLEAN,
        args: ARGUMENTLIST)
    ????
end proc;
```

Syntax

```
FunctionSignature → ParameterSignature ResultSignature

ParameterSignature → ( Parameters )
```

Parameters ⊡
«empty»
| *AllParameters*

AllParameters ⊡
Parameter
| *Parameter* **,** *AllParameters*
| *OptionalParameters*

OptionalParameters ⊡
OptionalParameter
| *OptionalParameter* **,** *OptionalParameters*
| *RestAndNamedParameters*

RestAndNamedParameters ⊡
NamedParameters
| *RestParameter*
| *RestParameter* **,** *NamedParameters*
| *NamedRestParameter*

NamedParameters ⊡
NamedParameter
| *NamedParameter* **,** *NamedParameters*

Parameter ⊡
TypedIdentifier^{allowIn}
| **const** *TypedIdentifier*^{allowIn}

OptionalParameter ⊡ *Parameter* **=** *AssignmentExpression*^{allowIn}

TypedInitialiser ⊡ *TypedIdentifier*^{allowIn} **=** *AssignmentExpression*^{allowIn}

NamedParameter ⊡
named *TypedInitialiser*
| **const named** *TypedInitialiser*
| **named const** *TypedInitialiser*

RestParameter ⊡
...
| **...** *Parameter*

NamedRestParameter ⊡
... named *Identifier*
| **... const named** *Identifier*
| **... named const** *Identifier*

ResultSignature ⊡
«empty»
| **:** *TypeExpression*^{allowIn}

Validation

proc Validate[*FunctionSignature* ⊡ *ParameterSignature ResultSignature*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
????
end proc;

Pre-Evaluation

```
proc PreEval[FunctionSignature ⇒ ParameterSignature ResultSignature]
        (cxt: CONTEXT, env: ENVIRONMENT, unchecked: BOOLEAN): SIGNATURE
    ????
end proc;

Untyped[FunctionSignature ⇒ ParameterSignature ResultSignature]: BOOLEAN = false;
```

15.5 Class Definition

Syntax

```
ClassDefinition ⇒ class Identifier Inheritance Block

Inheritance ⇒
      «empty»
    | extends TypeExpression^allowIn
```

Validation

```
Class[ClassDefinition]: CLASS;

proc Validate[ClassDefinition ⇒ class Identifier Inheritance Block]
        (cxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE)
    if pl ≠ singular then throw syntaxError end if;
    superclass: CLASS ⇒ Validate[Inheritance](cxt, env);
    a: COMPOUNDATTRIBUTE ⇒ toCompoundAttribute(attr);
    if not superclass.complete or superclass.final then throw definitionError end if;
    proc call(this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
        ????
    end proc;
    proc construct(args: ARGUMENTLIST, phase: PHASE): OBJECT
        ????
    end proc;
    prototype: OBJECT ⇒ null;
    if a.prototype then ???? end if;
    final: BOOLEAN;
    case a.memberMod of
        {none} do final ⇒ false;
        {static} do if env[0] ∈ CLASS then throw definitionError end if; final ⇒ false;
        {final} do final ⇒ true;
        {constructor, virtual} do throw definitionError
    end case;
    privateNamespace: NAMESPACE ⇒ new NAMESPACE[name: "private"];
    dynamic: BOOLEAN ⇒ a.dynamic or superclass.dynamic;
    c: CLASS ⇒ new CLASS[staticReadBindings: {}, staticWriteBindings: {}, instanceReadBindings: {},
            instanceWriteBindings: {}, instanceInitOrder: [], complete: false, super: superclass, prototype: prototype,
            privateNamespace: privateNamespace, dynamic: dynamic, allowNull: true, final: final, call: call,
            construct: construct];
    Class[ClassDefinition] ⇒ c;
    v: VARIABLE ⇒ new VARIABLE[type: classClass, value: c, immutable: true];
    defineStaticMember(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v);
    ValidateUsingFrame[Block](cxt, env, JUMPTARGETS[breakTargets: {}, continueTargets: {}], pl, c);
    c.complete ⇒ true
end proc;
```

```
proc Validate[Inheritance] (cxt: CONTEXT, env: ENVIRONMENT): CLASS
    [Inheritance ≡ «empty»] do return objectClass;
    [Inheritance ≡ extends TypeExpression^allowIn] do
        Validate[TypeExpression^allowIn](cxt, env);
        return Eval[TypeExpression^allowIn](env)
end proc;
```

Evaluation

```
proc Eval[ClassDefinition ≡ class Identifier Inheritance Block] (env: ENVIRONMENT, d: OBJECT): OBJECT
    c: CLASS ≡ Class[ClassDefinition];
    return EvalUsingFrame[Block](env, c, d)
end proc;
```

15.6 Namespace Definition

Syntax

```
NamespaceDefinition ≡ namespace Identifier
```

Validation

```
proc Validate[NamespaceDefinition ≡ namespace Identifier]
        (cxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE)
    if pl ≠ singular then throw syntaxError end if;
    a: COMPOUNDATTRIBUTE ≡ toCompoundAttribute(attr);
    if a.dynamic or a.prototype then throw definitionError end if;
    if not (a.memberMod = none or (a.memberMod = static and env[0] ≡ CLASS)) then
        throw definitionError
    end if;
    name: STRING ≡ Name[Identifier];
    ns: NAMESPACE ≡ new NAMESPACE⟦name: name⟧;
    v: VARIABLE ≡ new VARIABLE⟦type: namespaceClass, value: ns, immutable: true⟧;
    defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
end proc;
```

15.7 Package Definition

Syntax

```
PackageDefinition ≡
    package Block
  | package PackageName Block

PackageName ≡
    Identifier
  | PackageName . Identifier
```

16 Programs

Syntax

```
Program ≡ Directives
```

Evaluation

```
EvalProgram[Program □ Directives]: OBJECT
    begin
        savedPreEvaluators: (() □ ())[] □ preEvaluators;
        preEvaluators □ [];
        Validate[Directives](initialContext, initialEnvironment, JUMPTARGETS□breakTargets: {}, continueTargets: {}□
                singular, none);
        for each v □ preEvaluators do v() end for each;
        preEvaluators □ savedPreEvaluators;
        return Eval[Directives](initialEnvironment, undefined)
    end;
```

17 Predefined Identifiers

18 Built-in Classes

```
proc makeBuiltInClass(superclass: CLASSOPT, dynamic: BOOLEAN, allowNull: BOOLEAN, final: BOOLEAN): CLASS
    proc call(this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
        ????
    end proc;
    proc construct(args: ARGUMENTLIST, phase: PHASE): OBJECT
        ????
    end proc;
    privateNamespace: NAMESPACE □ new NAMESPACE□□name: "private"□□;
    return new CLASS□□staticReadBindings: {}, staticWriteBindings: {}, instanceReadBindings: {},
            instanceWriteBindings: {}, instanceInitOrder: [], complete: true, super: superclass, prototype: null,
            privateNamespace: privateNamespace, dynamic: dynamic, allowNull: allowNull, final: final, call: call,
            construct: construct□□
end proc;

objectClass: CLASS = makeBuiltInClass(none, false, true, false);

undefinedClass: CLASS = makeBuiltInClass(objectClass, false, false, true);

nullClass: CLASS = makeBuiltInClass(objectClass, false, true, true);

booleanClass: CLASS = makeBuiltInClass(objectClass, false, false, true);

generalNumberClass: CLASS = makeBuiltInClass(objectClass, false, false, false);

numberClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true);

longClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true);

uLongClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true);

characterClass: CLASS = makeBuiltInClass(objectClass, false, false, true);

stringClass: CLASS = makeBuiltInClass(objectClass, false, true, true);

namespaceClass: CLASS = makeBuiltInClass(objectClass, false, true, true);

attributeClass: CLASS = makeBuiltInClass(objectClass, false, true, true);

classClass: CLASS = makeBuiltInClass(objectClass, false, true, true);
```

functionClass: CLASS = *makeBuiltInClass*(*objectClass*, **false**, **true**, **true**);

prototypeClass: CLASS = *makeBuiltInClass*(*objectClass*, **true**, **true**, **true**);

packageClass: CLASS = *makeBuiltInClass*(*objectClass*, **true**, **true**, **true**);

objectPrototype: PROTOTYPE = **new** PROTOTYPE[[parent: **none**, dynamicProperties: {}]]

18.1 Object

18.2 Never

18.3 Void

18.4 Null

18.5 Boolean

18.6 Integer

18.7 Number

18.7.1 ToNumber Grammar

18.8 Character

18.9 String

18.10 Function

18.11 Array

18.12 Type

18.13 Math

18.14 Date

18.15 RegExp

18.15.1 Regular Expression Grammar

18.16 Unit

18.17 Error

18.18 Attribute

19 Built-in Functions

20 Built-in Attributes

~~21Built-in Operators~~

~~21.1Unary Operators~~

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

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

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

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

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

~~**proc** *callObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST, *phase*: PHASE): OBJECT~~
~~**case** *a* **of**~~
~~UNDEFINED | NULL | BOOLEAN | FLOAT64 | STRING | NAMESPACE | COMPOUNDATTRIBUTE | PROTOTYPE | PACKAGE | GLOBAL **do**~~
~~**throw badValueError**;~~
~~CLASS **do return** *a*.call(*this*, *args*, *phase*);~~
~~INSTANCE **do**~~
~~Note that *resolveAlias* is not called when getting the env field.~~
~~**return** *resolveAlias*(*a*).call(*this*, *args*, *a*.env, *phase*);~~
~~METHODCLOSURE **do**~~
~~*code*: {**abstract**} | INSTANCE ← *a*.method.code;~~
~~**case** *code* **of**~~
~~INSTANCE **do return** *callObject*(*a*.this, *code*, *args*, *phase*);~~
~~{**abstract**} **do throw propertyAccessError**~~
~~**end case**~~
~~**end case**~~
~~**end proc**;~~

~~**proc** *constructObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST, *phase*: PHASE): OBJECT~~
~~**case** *a* **of**~~
~~UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ METHODCLOSURE ∪ PROTOTYPE ∪ PACKAGE ∪ GLOBAL **do**~~
~~**throw badValueError**;~~
~~CLASS **do return** *a*.construct(*this*, *args*, *phase*);~~
~~INSTANCE **do**~~
~~Note that *resolveAlias* is not called when getting the env field.~~
~~**return** *resolveAlias*(*a*).construct(*this*, *args*, *a*.env, *phase*)~~
~~**end case**~~
~~**end proc**;~~

~~**proc** *bracketReadObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST, *phase*: PHASE): OBJECT~~
~~**if** |*args*.positional| ≠ 1 **or** *args*.named ≠ {} **then throw argumentMismatchError end if**;~~
~~*name*: STRING ← *toString*(*args*.positional[0], *phase*);~~
~~*result*: OBJECTOPT ← *readProperty*(*a*, {QUALIFIEDNAME namespace: *publicNamespace*, id: *name*},~~
~~**propertyLookup**, *phase*);~~
~~**if** *result* = **none then throw propertyAccessError else return** *result* **end if**~~
~~**end proc**;~~

~~**proc** *bracketWriteObject*(*this*: OBJECT, *a*: OBJECT, *args*: ARGUMENTLIST, *phase*: PHASE): OBJECT~~
~~**if** *phase* = **compile then throw compileExpressionError end if**;~~
~~**if** |*args*.positional| ≠ 2 **or** *args*.named ≠ {} **then throw argumentMismatchError end if**;~~
~~*newValue*: OBJECT ← *args*.positional[0];~~
~~*name*: STRING ← *toString*(*args*.positional[1], *phase*);~~
~~*result*: {**none**, **ok**} ← *writeProperty*(*a*, {QUALIFIEDNAME namespace: *publicNamespace*, id: *name*},~~
~~**propertyLookup**, **true**, *newValue*, *phase*);~~
~~**if** *result* = **none then throw propertyAccessError end if**;~~
~~**return undefined**~~
~~**end proc**;~~

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

~~*plusTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *plusObject*};~~

~~*minusTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *minusObject*};~~

~~*bitwiseNotTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *bitwiseNotObject*};~~

~~*incrementTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *incrementObject*};~~

~~*decrementTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *decrementObject*};~~

~~*callTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *callObject*};~~

~~*constructTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *constructObject*};~~

~~*bracketReadTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *bracketReadObject*};~~

~~*bracketWriteTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *bracketWriteObject*};~~

~~*bracketDeleteTable*: UNARYMETHOD{} ← {UNARYMETHOD operandType: *objectClass*, f: *bracketDeleteObject*};~~

~~21.2Binary Operators~~

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

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

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

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

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

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

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

```
proc equalObjects(a: OBJECT, b: OBJECT, phase: PHASE): 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, phase), b, phase)
            end if;
        FLOAT64 do
            bp: PRIMITIVEOBJECT | toPrimitive(b, null, phase);
            case bp of
                UNDEFINED | NULL do return false;
                BOOLEAN | FLOAT64 | STRING do
                    return float64Compare(a, toNumber(bp, phase)) = equal
            end case;
        STRING do
            bp: PRIMITIVEOBJECT | toPrimitive(b, null, phase);
            case bp of
                UNDEFINED | NULL do return false;
                BOOLEAN | FLOAT64 do
                    return float64Compare(toNumber(a, phase), toNumber(bp, phase)) = equal;
                STRING do return a = bp
            end case;
        NAMESPACE | COMPOUNDATTRIBUTE | CLASS | METHODCLOSURE | PROTOTYPE | INSTANCE | PACKAGE |
                GLOBAL do
            case b of
                UNDEFINED | NULL do return false;
                NAMESPACE | COMPOUNDATTRIBUTE | CLASS | METHODCLOSURE | PROTOTYPE | INSTANCE |
                        PACKAGE | GLOBAL do
                    return strictEqualObjects(a, b, phase);
                BOOLEAN | FLOAT64 | STRING do
                    ap: PRIMITIVEOBJECT | toPrimitive(a, null, phase);
                    case ap of
                        UNDEFINED | NULL do return false;
                        BOOLEAN | FLOAT64 | STRING do return equalObjects(ap, b, phase)
                    end case
            end case
    end case
end proc;

proc strictEqualObjects(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    if a | ALIASINSTANCE then return strictEqualObjects(a.original, b, phase)
    elsif b | ALIASINSTANCE then return strictEqualObjects(a, b.original, phase)
    elsif 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, phase: PHASE): OBJECT
    i: INTEGER | toUInt32(toNumber(a, phase));
    count: INTEGER | bitwiseAnd(toUInt32(toNumber(b, phase)), 0x1F);
    return realToFloat64(uInt32ToInt32(bitwiseAnd(bitwiseShift(i, count), 0xFFFFFFFF)))
end proc;
```

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

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

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

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

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

~~*addTable*: BINARYMETHOD{} ← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *addObjects*]};~~

~~*subtractTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *subtractObjects*]};~~

~~*multiplyTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *multiplyObjects*]};~~

~~*divideTable*: BINARYMETHOD{} ← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *divideObjects*]};~~

~~*remainderTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *remainderObjects*]};~~

~~*lessTable*: BINARYMETHOD{} ← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *lessObjects*]};~~

~~*lessOrEqualTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *lessOrEqualObjects*]};~~

~~*equalTable*: BINARYMETHOD{} ← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *equalObjects*]};~~

~~*strictEqualTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *strictEqualObjects*]};~~

~~*shiftLeftTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *shiftLeftObjects*]};~~

~~*shiftRightTable*: BINARYMETHOD{}~~
~~← {BINARYMETHOD[leftType: *objectClass*, rightType: *objectClass*, f: *shiftRightObjects*]};~~

~~*shiftRightUnsignedTable*: BINARYMETHOD{} - {BINARYMETHOD leftType: *objectClass*, rightType: *objectClass*, f: *shiftRightUnsignedObjects* ;~~

~~*bitwiseAndTable*: BINARYMETHOD{}~~
~~- {BINARYMETHOD leftType: *objectClass*, rightType: *objectClass*, f: *bitwiseAndObjects* ;~~

~~*bitwiseXorTable*: BINARYMETHOD{}~~
~~- {BINARYMETHOD leftType: *objectClass*, rightType: *objectClass*, f: *bitwiseXorObjects* ;~~

~~*bitwiseOrTable*: BINARYMETHOD{}~~
~~- {BINARYMETHOD leftType: *objectClass*, rightType: *objectClass*, f: *bitwiseOrObjects* ;~~

~~22~~21 Built-in Namespaces

~~23Built-in Units~~

~~24~~22 Errors

~~25~~23 Optional Packages

~~25.1~~23.1 Machine Types

~~25.2~~23.2 Internationalisation

~~25.3Units~~

A Index

A.1 Nonterminals

A.2 Tags

A.3 Semantic Domains

A.4 Globals