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.

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

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

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

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

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

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

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:

$x \in A$	**true** if *x* is an element of *A* and **false** if not
$x \notin A$	**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 $m \in A$ and for all elements $x \in A$, $x \ge m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
max *A*	The value *m* that satisfies both $m \in A$ and for all elements $x \in A$, $x \le m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
$A \cap B$	The intersection of *A* and *B* (the set or semantic domain of all values that are present both in *A* and in *B*)
$A \cup B$	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*.
$A \ne B$	**false** if *A* and *B* are equal and **true** otherwise
$A \subseteq B$	**true** if *A* is a subset of *B* and **false** otherwise. *A* is a subset of *B* if every element of *A* is also in *B*. Every set is a subset of itself. The empty set {} is a subset of every set.
$A \subset B$	**true** if *A* is a proper subset of *B* and **false** otherwise. $A \subset B$ is equivalent to $A \subseteq B$ **and** $A \ne B$.

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

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

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

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

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

The quantifier

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

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

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

5.6 Real Numbers

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

INTEGER is the semantic domain 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 $i \leq x < i+1$. $\lfloor \pi \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, and $\lfloor 7 \rfloor = 7$.
$\lceil x \rceil$	*Ceiling* of x, which is the unique integer i such that $i-1 < x \leq i$. $\lceil \pi \rceil = 4$, $\lceil -3.5 \rceil = -3$, and $\lceil 7 \rceil = 7$.
x **mod** y	x modulo y, which is defined as $x - y\lfloor x/y \rfloor$ y must not be zero. 10 **mod** 7 = 3, and –1 **mod** 7 = 6.

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

5.6.1 Bitwise Integer Operators

The four procedures below perform bitwise operations on integers. The integers are treated as though they were written in infinite-precision two’s complement binary notation, with each 1 bit representing **true** and 0 bit representing **false**.

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

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

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

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

5.7 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**.

The procedures *characterToCode* and *codeToCharacter* convert between characters and their integer Unicode values.

characterToCode(*c*: CHARACTER): {0 ... 65535}	Return character *c*’s Unicode code point as an integer
codeToCharacter(*i*: {0 ... 65535}): CHARACTER	Return the character whose Unicode code point is *i*

5.8 Lists

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

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

For example, the following list contains four strings:

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

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

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

A list can also be written using the list comprehension notation

[$f(x)$ | $\forall x \in u$**]**

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

[$f(x)$ | $\forall x \in u$ **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,

$[x^2 \mid \forall x \in [-1, 1, 2, 3, 4, 2, 5]] = [1, 1, 4, 9, 16, 4, 25]$
$[x+1 \mid \forall x \in [-1, 1, 2, 3, 4, 5, 3, 10]$ **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, e be an element, 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 $[e_0, e_1, \ldots, e_{n-1}, f_0, f_1, \ldots, f_{m-1}]$
repeat(e, i)	$i \ge 0$	The list $[e, e, \ldots, e]$ of length i containing i identical elements e
$u = v$		**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 $x \in u$ **satisfies** *predicate*(x)
every $x \in u$ **satisfies** *predicate*(x)

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

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

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

5.9 Strings

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

"`Wonder`«LF»"

is equivalent to:

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

The empty string is usually written as "".

In addition to 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[].

5.10 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.

5.11 Records

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

A record is comprised of a 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 *i*th 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.

5.12 ECMAScript Numeric Types

ECMAScript does not support exact real numbers as one of the programmer-visible data types. Instead, ECMAScript numbers have finite range and precision. The semantic domain of all programmer-visible numbers representable in ECMAScript is GENERALNUMBER, defined as the union of four basic numeric semantic domains LONG, ULONG, FLOAT32, and FLOAT64:

GENERALNUMBER = LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64

The four basic numeric semantic domains are all disjoint from each other and from the semantic domains INTEGER, RATIONAL, and REAL.

The semantic domain FINITEGENERALNUMBER is the subtype of all finite values in GENERALNUMBER:

FINITEGENERALNUMBER = LONG ∪ ULONG ∪ FINITEFLOAT32 ∪ FINITEFLOAT64

5.12.1 Signed Long Integers

Programmer-visible signed 64-bit long integers are represented by the semantic domain LONG. These are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains ULONG, FLOAT32, and FLOAT64. A LONG tuple has the field below:

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

5.12.2 Unsigned Long Integers

Programmer-visible unsigned 64-bit long integers are represented by the semantic domain ULONG. These are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains LONG, FLOAT32, and FLOAT64. A ULONG tuple has the field below:

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

5.12.3 Single-Precision Floating-Point Numbers

FLOAT32 is the semantic domain of all representable single-precision floating-point IEEE 754 values, with all not-a-number values considered indistinguishable from each other. FLOAT32 is the union of the following semantic domains:

FLOAT32 = FINITEFLOAT32 ∪ {**+∞**$_{\mathbf{f32}}$, **–∞**$_{\mathbf{f32}}$, **NaN**$_{\mathbf{f32}}$};
FINITEFLOAT32 = NONZEROFINITEFLOAT32 ∪ {**+zero**$_{\mathbf{f32}}$, **–zero**$_{\mathbf{f32}}$}

The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains LONG, ULONG, and FLOAT64. A NONZEROFINITEFLOAT32 tuple has the field below:

Field	Contents	Note
value	NORMALISEDFLOAT32VALUES ∪ DENORMALISEDFLOAT32VALUES	The value, represented as an exact rational number

There are 4261412864 (that is, $2^{32}–2^{25}$) *normalised* values:

NORMALISEDFLOAT32VALUES = $\{s \times m \times 2^{e} \mid \forall s \in \{–1, 1\}, \forall m \in \{2^{23} \ldots 2^{24}–1\}, \forall e \in \{–149 \ldots 104\}\}$

m is called the significand.

There are also 16777214 (that is, $2^{24}–2$) *denormalised* non-zero values:

DENORMALISEDFLOAT32VALUES = $\{s \times m \times 2^{–149} \mid \forall s \in \{–1, 1\}, \forall m \in \{1 \ldots 2^{23}–1\}\}$

m is called the significand.

The remaining FLOAT32 values are the tags **+zero**$_{\mathbf{f32}}$ (positive zero), **–zero**$_{\mathbf{f32}}$ (negative zero), **+∞**$_{\mathbf{f32}}$ (positive infinity), **–∞**$_{\mathbf{f32}}$ (negative infinity), and **NaN**$_{\mathbf{f32}}$ (not a number).

Members of the semantic domain NONZEROFINITEFLOAT32 with value greater than zero are called *positive finite*. The remaining members of NONZEROFINITEFLOAT32 are called *negative finite*.

Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation = and ≠ may be used to compare them. Note that = is **false** for different tags, so **+zero**$_{\mathbf{f32}}$ ≠ **–zero**$_{\mathbf{f32}}$ but **NaN**$_{\mathbf{f32}}$ = **NaN**$_{\mathbf{f32}}$. The ECMAScript *x* == *y* and *x* === *y* operators have different behavior for FLOAT32 values, defined by *isEqual* and *isStrictlyEqual*.

5.12.3.1 Shorthand Notation

In this specification, when *x* is a real number or expression, the notation $x_{\mathbf{f32}}$ indicates the result of *realToFloat32*(*x*), which is the "closest" FLOAT32 value as defined below. Thus, 3.4 is a REAL number, while $3.4_{\mathbf{f32}}$ is a FLOAT32 value (whose exact value is actually 3.400000095367431640625). The positive finite FLOAT32 values range from $10^{–45}{}_{\mathbf{f32}}$ to $(3.4028235 \times 10^{38})_{\mathbf{f32}}$.

5.12.3.2 Conversion

The procedure *realToFloat32* converts a real number *x* into the applicable element of FLOAT32 as follows:

```
proc realToFloat32(x: REAL): FLOAT32
    s: RATIONAL{} ← NORMALISEDFLOAT32VALUES ∪ DENORMALISEDFLOAT32VALUES ∪ {$–2^{128}$, 0, $2^{128}$};
    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 $–2^{128}$, 0, and $2^{128}$ are considered to have
            even significands.
    if a = $2^{128}$ then return $+\infty_{f32}$
    elsif a = $–2^{128}$ then return $–\infty_{f32}$
    elsif a ≠ 0 then return NONZEROFINITEFLOAT32⟨⟨value: a⟩⟩
    elsif x < 0 then return $–zero_{f32}$
    else return $+zero_{f32}$
    end if
end proc
```

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

The procedure *truncateFiniteFloat32* truncates a FINITEFLOAT32 value to an integer, rounding towards zero:

```
proc truncateFiniteFloat32(x: FINITEFLOAT32): INTEGER
    if x ∈ {$+zero_{f32}$, $–zero_{f32}$} then return 0 end if;
    r: RATIONAL ← x.value;
    if r > 0 then return ⌊r⌋ else return ⌈r⌉ end if
end proc
```

5.12.3.3 Arithmetic

The following table defines negation of FLOAT32 values using IEEE 754 rules. Note that $(expr)_{\mathbf{f32}}$ is a shorthand for *realToFloat32*(*expr*).

float32Negate(*x*: FLOAT32): FLOAT32

x	Result
$-\infty_{\mathbf{f32}}$	$+\infty_{\mathbf{f32}}$
negative finite	$(-x.\text{value})_{\mathbf{f32}}$
$\textbf{–zero}_{\mathbf{f32}}$	$\textbf{+zero}_{\mathbf{f32}}$
$\textbf{+zero}_{\mathbf{f32}}$	$\textbf{–zero}_{\mathbf{f32}}$
positive finite	$(-x.\text{value})_{\mathbf{f32}}$
$+\infty_{\mathbf{f32}}$	$-\infty_{\mathbf{f32}}$
$\textbf{NaN}_{\mathbf{f32}}$	$\textbf{NaN}_{\mathbf{f32}}$

5.12.4 Double-Precision Floating-Point Numbers

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

FLOAT64 = FINITEFLOAT64 □ $\{+\infty_{\mathbf{f64}}, -\infty_{\mathbf{f64}}, \textbf{NaN}_{\mathbf{f64}}\}$;
FINITEFLOAT64 = NONZEROFINITEFLOAT64 □ $\{\textbf{+zero}_{\mathbf{f64}}, \textbf{–zero}_{\mathbf{f64}}\}$

The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains LONG, ULONG, and FLOAT32. A NONZEROFINITEFLOAT64 tuple has the field below:

Field	Contents	Note
value	NORMALISEDFLOAT64VALUES □ DENORMALISEDFLOAT64VALUES	The value, represented as an exact rational number

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

NORMALISEDFLOAT64VALUES = $\{s□m□2^{e} \mid □s □ \{–1, 1\}, □m □ \{2^{52} \ldots 2^{53}–1\}, □e □ \{–1074 \ldots 971\}\}$

m is called the significand.

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

DENORMALISEDFLOAT64VALUES = $\{s□m□2^{–1074} \mid □s □ \{–1, 1\}, □m □ \{1 \ldots 2^{52}–1\}\}$

m is called the significand.

The remaining FLOAT64 values are the tags $\textbf{+zero}_{\mathbf{f64}}$ (positive zero), $\textbf{–zero}_{\mathbf{f64}}$ (negative zero), $+\infty_{\mathbf{f64}}$ (positive infinity), $-\infty_{\mathbf{f64}}$ (negative infinity), and $\textbf{NaN}_{\mathbf{f64}}$ (not a number).

Members of the semantic domain NONZEROFINITEFLOAT64 with value greater than zero are called *positive finite*. The remaining members of NONZEROFINITEFLOAT64 are called *negative finite*.

Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation = and ≠ may be used to compare them. Note that = is **false** for different tags, so $\textbf{+zero}_{\mathbf{f64}} \neq \textbf{–zero}_{\mathbf{f64}}$ but $\textbf{NaN}_{\mathbf{f64}} = \textbf{NaN}_{\mathbf{f64}}$. The ECMAScript *x* == *y* and *x* === *y* operators have different behavior for FLOAT64 values, defined by *isEqual* and *isStrictlyEqual*.

5.12.4.1 Shorthand Notation

In this specification, when *x* is a real number or expression, the notation $x_{\mathbf{f64}}$ indicates the result of *realToFloat64*(*x*), which is the "closest" FLOAT64 value as defined below. Thus, 3.4 is a REAL number, while $3.4_{\mathbf{f64}}$ is a FLOAT64 value (whose exact value is actually 3.399999999999999911182158029987476766109466552734375). The positive finite FLOAT64 values range from $(5 □ 10^{–324})_{\mathbf{f64}}$ to $(1.7976931348623157 □ 10^{308})_{\mathbf{f64}}$.

5.12.4.2 Conversion

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

```
proc realToFloat64(x: REAL): FLOAT64
   s: RATIONAL{} □ NORMALISEDFLOAT64VALUES □ DENORMALISEDFLOAT64VALUES □ {$-2^{1024}$, 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 $+\infty_{f64}$
   elsif $a = -2^{1024}$ then return $-\infty_{f64}$
   elsif $a \neq 0$ then return NONZEROFINITEFLOAT64□value: a□
   elsif $x < 0$ then return $-\text{zero}_{f64}$
   else return $+\text{zero}_{f64}$
   end if
end proc
```

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

The procedure *float32ToFloat64* converts a FLOAT32 number *x* into the corresponding FLOAT64 number as defined by the following table:

float32ToFloat64(*x*: FLOAT32): FLOAT64

x	**Result**
$-\infty_{f32}$	$-\infty_{f64}$
$-\text{zero}_{f32}$	$-\text{zero}_{f64}$
$+\text{zero}_{f32}$	$+\text{zero}_{f64}$
$+\infty_{f32}$	$+\infty_{f64}$
NaN_{f32}	NaN_{f64}
Any NONZEROFINITEFLOAT32 value	NONZEROFINITEFLOAT64□value: ***x*.value**□

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

```
proc truncateFiniteFloat64(x: FINITEFLOAT64): INTEGER
   if x □ {$+\text{zero}_{f64}$, $-\text{zero}_{f64}$} then return 0 end if;
   r: RATIONAL □ x.value;
   if $r > 0$ then return □r□ else return □r□ end if
end proc
```

5.12.4.3 Arithmetic

The following tables define procedures that perform common arithmetic on FLOAT64 values using IEEE 754 rules. Note that $(expr)_{f64}$ is a shorthand for *realToFloat64*(*expr*).

float64Abs(*x*: FLOAT64): FLOAT64

x	**Result**
$-\infty_{f64}$	$+\infty_{f64}$
negative finite	$(-x.\text{value})_{f64}$
$-\text{zero}_{f64}$	$+\text{zero}_{f64}$
$+\text{zero}_{f64}$	$+\text{zero}_{f64}$
positive finite	*x*
$+\infty_{f64}$	$+\infty_{f64}$
NaN_{f64}	NaN_{f64}

float64Negate(x: FLOAT64): FLOAT64

x	**Result**
$-\infty_{f64}$	$+\infty_{f64}$
negative finite	$(-x.\text{value})_{f64}$
$-\text{zero}_{f64}$	$+\text{zero}_{f64}$
$+\text{zero}_{f64}$	$-\text{zero}_{f64}$
positive finite	$(-x.\text{value})_{f64}$
$+\infty_{f64}$	$-\infty_{f64}$
NaN_{f64}	NaN_{f64}

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

	y						
x	$-\infty_{f64}$	negative finite	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}	NaN_{f64}
negative finite	$-\infty_{f64}$	$(x.\text{value} + y.\text{value})_{f64}$	x	x	$(x.\text{value} + y.\text{value})_{f64}$	$+\infty_{f64}$	NaN_{f64}
$-\text{zero}_{f64}$	$-\infty_{f64}$	y	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	y	$+\infty_{f64}$	NaN_{f64}
$+\text{zero}_{f64}$	$-\infty_{f64}$	y	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	y	$+\infty_{f64}$	NaN_{f64}
positive finite	$-\infty_{f64}$	$(x.\text{value} + y.\text{value})_{f64}$	x	x	$(x.\text{value} + y.\text{value})_{f64}$	$+\infty_{f64}$	NaN_{f64}
$+\infty_{f64}$	NaN_{f64}	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

NOTE The identity for floating-point addition is $-\text{zero}_{f64}$, not $+\text{zero}_{f64}$.

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

	y						
x	$-\infty_{f64}$	negative finite	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	NaN_{f64}	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}
negative finite	$+\infty_{f64}$	$(x.\text{value} - y.\text{value})_{f64}$	x	x	$(x.\text{value} - y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
$-\text{zero}_{f64}$	$+\infty_{f64}$	$(-y.\text{value})_{f64}$	$+\text{zero}_{f64}$	$-\text{zero}_{f64}$	$(-y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
$+\text{zero}_{f64}$	$+\infty_{f64}$	$(-y.\text{value})_{f64}$	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	$(-y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
positive finite	$+\infty_{f64}$	$(x.\text{value} - y.\text{value})_{f64}$	x	x	$(x.\text{value} - y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

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

	y						
x	$-\infty_{f64}$	negative finite	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}	NaN_{f64}	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}
negative finite	$+\infty_{f64}$	$(x.\text{value} \,\square\, y.\text{value})_{f64}$	$+\text{zero}_{f64}$	$-\text{zero}_{f64}$	$(x.\text{value} \,\square\, y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
$-\text{zero}_{f64}$	NaN_{f64}	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}
$+\text{zero}_{f64}$	NaN_{f64}	$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}
positive finite	$-\infty_{f64}$	$(x.\text{value} \,\square\, y.\text{value})_{f64}$	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	$(x.\text{value} \,\square\, y.\text{value})_{f64}$	$+\infty_{f64}$	NaN_{f64}
$+\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}	NaN_{f64}	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

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

x \ *y*	$-\infty_{f64}$	negative finite	$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	NaN_{f64}	$+\infty_{f64}$	$+\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}	NaN_{f64}
negative finite	$+\text{zero}_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$+\infty_{f64}$	$-\infty_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$-\text{zero}_{f64}$	NaN_{f64}
$-\text{zero}_{f64}$	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}	$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	NaN_{f64}
$+\text{zero}_{f64}$	$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	NaN_{f64}
positive finite	$-\text{zero}_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$-\infty_{f64}$	$+\infty_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$+\text{zero}_{f64}$	NaN_{f64}
$+\infty_{f64}$	NaN_{f64}	$-\infty_{f64}$	$-\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

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

x \ *y*	$-\infty_{f64}$, $+\infty_{f64}$	positive or negative finite	$-\text{zero}_{f64}$, $+\text{zero}_{f64}$	NaN_{f64}
$-\infty_{f64}$	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}
negative finite	*x*	*float64Negate*(*float64Remainder*(*float64Negate*(*x*), *y*))	NaN_{f64}	NaN_{f64}
$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	$-\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}
$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	$+\text{zero}_{f64}$	NaN_{f64}	NaN_{f64}
positive finite	*x*	$(x.\text{value} - \lvert y.\text{value}\rvert □□ x.\text{value}/\lvert y.\text{value}\rvert □)_{f64}$	NaN_{f64}	NaN_{f64}
$+\infty_{f64}$	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

Note that *float64Remainder*(*float64Negate*(*x*), *y*) always produces the same result as *float64Negate*(*float64Remainder*(*x*, *y*)). Also, *float64Remainder*(*x*, *float64Negate*(*y*)) always produces the same result as *float64Remainder*(*x*, *y*).

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(param₁: T₁, … , paramₙ: Tₙ): T
   step₁;
   step₂;
   … ;
   stepₘ
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 □ T_2 □ ... □ T_n □ T. If *n* = 0, this semantic domain is written as () □ T. If the procedure does not produce a result, the semantic domain of procedures is written either as T_1 □ T_2 □ ... □ T_n □ () or as () □ ().

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.

note *Comment*

A **note** step performs no operation. It provides an informative comment about the algorithm. If *Comment* is an expression, then the **note** step is an informative comment that asserts that the expression, if evaluated at this point, would be guaranteed to evaluate to **true**.

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
   T1 do step; step; ...; step;
   T2 do step; step; ...; step;
   ...;
   Tn 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 ∈ 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 α that contains a nonterminal *N*, one may replace an occurrence of *N* in α with the right-hand side of any production for which *N* is the left-hand side. The resulting sequence of grammar symbols is also a sentential form.

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

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

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

5.14.1 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*$^{\text{allowIn}}$

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

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

5.14.4 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*$^{\text{normal},□}$
| *LeftSideExpression*$^{□}$ *CompoundAssignment* *AssignmentExpression*$^{\text{normal},□}$

expands into the following four rules:

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

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

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

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

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

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

5.14.5 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** * | /

5.15 Semantic Actions

Semantic actions tie the grammar and the semantics together. A semantic action ascribes semantic meaning to a grammar production.

Two examples illustrates the use of semantic actions. A description of the notation for specifying semantic actions follows the examples.

5.15.1 Example

Consider the following sample grammar, with the start nonterminal *Numeral*:

```
Digit □  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Digits □
      Digit
   |  Digits Digit

Numeral □
      Digits
   |  Digits # Digits
```

This grammar defines the syntax of an acceptable input: “`37`”, “`33#4`” and “`30#2`” are acceptable syntactically, while “`1a`” is not. However, the grammar does not indicate what these various inputs mean. That is the function of the semantics, which are defined in terms of actions on the parse tree of grammar rule expansions. Consider the following sample set of actions defined on this grammar, with a starting *Numeral* action called (in this example) Value:

```
Value[Digit]: INTEGER = Digit’s decimal value (an integer between 0 and 9).

DecimalValue[Digits]: INTEGER;
   DecimalValue[Digits □  Digit] = Value[Digit];
   DecimalValue[Digits$_0$ □  Digits$_1$ Digit] = 10□DecimalValue[Digits$_1$] + Value[Digit];

proc BaseValue[Digits] (base: INTEGER): INTEGER
   [Digits □  Digit] do
      d: INTEGER □  Value[Digit];
      if $d < base$ then return d else throw syntaxError end if;
   [Digits$_0$ □  Digits$_1$ Digit] do
      d: INTEGER □  Value[Digit];
      if $d < base$ then return base□BaseValue[Digits$_1$](base) + d
      else throw syntaxError
      end if
end proc;

Value[Numeral]: INTEGER;
   Value[Numeral □  Digits] = DecimalValue[Digits];
   Value[Numeral □  Digits$_1$ # Digits$_2$]
      begin
         base: INTEGER □  DecimalValue[Digits$_2$];
         if $base \geq 2$ and $base \leq 10$ then return BaseValue[Digits$_1$](base)
         else throw syntaxError
         end if
      end;
```

Action names are written in cursive type. The definition

Value[*Numeral*]: INTEGER;

states that the action Value can be applied to any expansion of the nonterminal *Numeral*, and the result is an INTEGER. This action either maps an input to an integer or throws an exception. The code above throws the exception **syntaxError** when presented with the input “`30#2`”.

There are two definitions of the Value action on *Numeral*, one for each grammar production that expands *Numeral*:

```
Value[Numeral → Digits] = DecimalValue[Digits];
Value[Numeral → $Digits_1$ # $Digits_2$]
  begin
    base: INTEGER ← DecimalValue[$Digits_2$];
    if base $\geq$ 2 and base $\leq$ 10 then return BaseValue[$Digits_1$](base)
    else throw syntaxError
    end if
  end;
```

Each definition of an action is allowed to perform actions on the terminals and nonterminals on the right side of the expansion. For example, Value applied to the first *Numeral* production (the one that expands *Numeral* into *Digits*) simply applies the DecimalValue action to the expansion of the nonterminal *Digits* and returns the result. On the other hand, Value applied to the second *Numeral* production (the one that expands *Numeral* into *Digits* # *Digits*) performs a computation using the results of the DecimalValue and BaseValue applied to the two expansions of the *Digits* nonterminals. In this case there are two identical nonterminals *Digits* on the right side of the expansion, so subscripts are used to indicate on which the actions DecimalValue and BaseValue are performed.

The definition

```
proc BaseValue[Digits] (base: INTEGER): INTEGER
  [Digits → Digit] do
    d: INTEGER ← Value[Digit];
    if d < base then return d else throw syntaxError end if;
  [$Digits_0$ → $Digits_1$ Digit] do
    d: INTEGER ← Value[Digit];
    if d < base then return base×BaseValue[$Digits_1$](base) + d
    else throw syntaxError
    end if
end proc;
```

states that the action BaseValue can be applied to any expansion of the nonterminal *Digits*, and the result is a procedure that takes one INTEGER argument *base* and returns an INTEGER. The procedure's body is comprised of independent cases for each production that expands *Digits*. When the procedure is called, the case corresponding to the expansion of the nonterminal *Digits* is evaluated.

The Value action on *Digit*

Value[*Digit*]: INTEGER = *Digit*'s decimal value (an integer between 0 and 9)

illustrates the direct use of a nonterminal *Digit* in a semantic expression. Using the nonterminal *Digit* in this way refers to the character into which the *Digit* grammar rule expands.

The semantics can be evaluated on the sample inputs to get the following results:

Input	Semantic Result
`37`	37
`33#4`	15
`30#2`	**throw syntaxError**

5.15.2 Abbreviated Actions

In some cases the all actions named A for a nonterminal *N*'s rule are repetitive, merely calling A on the nonterminals on the right side of the expansions of *N* in the grammar. In these cases the semantics of action A are abbreviated, as illustrated by the example below.

Given the sample grammar rule

```
Expression □
      Subexpression
   |  Expression * Subexpression
   |  Subexpression + Subexpression
   |  this
```

the notation

Validate[*Expression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *Expression*.

is an abbreviation for the following:

```
proc Validate[Expression] (cxt: CONTEXT, env: ENVIRONMENT)
   [Expression □ Subexpression] do Validate[Subexpression](cxt, env);
   [$Expression_0$ □ $Expression_1$ * Subexpression] do
      Validate[$Expression_1$](cxt, env);
      Validate[Subexpression](cxt, env);
   [Expression □ $Subexpression_1$ + $Subexpression_2$] do
      Validate[$Subexpression_1$](cxt, env);
      Validate[$Subexpression_2$](cxt, env);
   [Expression □ this] do nothing
end proc;
```

Note that:

- The expanded calls to Validate get the same arguments *cxt* and *env* passed in to the call to Validate on *Expression*.
- When an expansion of *Expression* has more than one nonterminal on its right side, Validate is called on all of the nonterminals in left-to-right order.
- When an expansion of *Expression* has no nonterminals on its right side, Validate does nothing.

5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

Action[*nonterminal*]: T;

This notation states that action Action can be performed on nonterminal *nonterminal* and returns a value that is a member of the semantic domain T. The action's value is either defined using the notation Action[*nonterminal* □ *expansion*] = *expression* below or set as a side effect of computing another action via an action assignment.

Action[*nonterminal* □ *expansion*] = *expression*;

This notation specifies the value that action Action on nonterminal *nonterminal* computes in the case where nonterminal *nonterminal* expands to the given *expansion*. *expansion* can contain zero or more terminals and nonterminals (as well as other notations allowed on the right side of a grammar production). Furthermore, the terminals and nonterminals of *expansion* can be subscripted to allow them to be unambiguously referenced by action references or nonterminal references inside *expression*.

Action[*nonterminal* □ *expansion*]: T = *expression*;

This notation combines the above two — it specifies the semantic domain of the action as well as its value.

```
Action[nonterminal □ expansion]
   begin
      $step_1$;
      $step_2$;
      … ;
      $step_m$
   end;
```

This notation is used when the computation of the action is too complex for an expression. Here the steps to compute the action are listed as *step*$_1$ through *step*$_m$. A **return** step produces the value of the action.

```
proc Action[nonterminal □ expansion] (param$_1$: T$_1$, ... , param$_n$: T$_n$): T
   step$_1$;
   step$_2$;
   ... ;
   step$_m$
end proc;
```

This notation is used only when Action returns a procedure when applied to nonterminal *nonterminal* with a single expansion *expansion*. Here the steps of the procedure are listed as *step*$_1$ through *step*$_m$.

```
proc Action[nonterminal] (param$_1$: T$_1$, ... , param$_n$: T$_n$): T
   [nonterminal □ expansion$_1$] do
      step;
      ... ;
      step;
   [nonterminal □ expansion$_2$] do
      step;
      ... ;
      step;
   ...;
   [nonterminal □ expansion$_n$] do
      step;
      ... ;
      step
end proc;
```

This notation is used only when Action returns a procedure when applied to nonterminal *nonterminal* with several expansions *expansion*$_1$ through *expansion*$_n$. The procedure is comprised of a series of cases, one for each expansion. Only the steps corresponding to the expansion found by the grammar parser used are evaluated.

Action[*nonterminal*] (*param*$_1$: T$_1$, ... , *param*$_n$: T$_n$) propagates the call to Action to every nonterminal in the expansion of *nonterminal*.

This notation is an abbreviation stating that calling Action on *nonterminal* causes Action to be called with the same arguments on every nonterminal on the right side of the appropriate expansion of *nonterminal*. See section 5.15.2.

5.16 Other Semantic Definitions

In addition to actions (section 5.15.3), the semantics sometimes define supporting top-level procedures and variables. The following notation is used for these definitions:

name: T = *expression*;

This notation defines *name* to be a constant value given by the result of computing *expression*. The value is guaranteed to be a member of the semantic domain T.

name: T □ *expression*;

This notation defines *name* to be a mutable global value. Its initial value is the result of computing *expression*, but it may be subsequently altered using an assignment. The value is guaranteed to be a member of the semantic domain T.

```
proc f(param$_1$: T$_1$, ... , param$_n$: T$_n$): T
   step$_1$;
   step$_2$;
   ... ;
   step$_m$
end proc;
```

This notation defines *f* to be a procedure (section 5.13).

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 currently used by ECMAScript `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `default`, `delete`, `do`, `else`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `is`, `namespace`, `new`, `null`, `package`, `private`, `public`, `return`, `static`, `super`, `switch`, `this`, `throw`, `true`, `try`, `typeof`, `use`, `var`, `void`, `while`, `with`.
  - One of the reserved words reserved for future use `abstract`, `debugger`, `enum`, `goto`, `implements`, `interface`, `native`, `protected`, `synchronized`, `throws`, `transient`, `volatile`.

- 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 GENERALNUMBER that is the number's value.
- A **NegatedMinLong** token, which carries no value. This token is the result of evaluating `9223372036854775808L`.
- 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.

TOKEN is the semantic domain of all tokens. INPUTELEMENT is the semantic domain of all input elements, and is defined by:

INPUTELEMENT = {**LineBreak**, **EndOfInput**} ∪ TOKEN

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 $NextInputElement^{\mathrm{re}}$, $NextInputElement^{\mathrm{div}}$, and $NextInputElement^{\mathrm{num}}$, 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 $NextInputElement^{\mathrm{num}}$ if the previous lexed token was a **Number** or **NegatedMinLong**, $NextInputElement^{\mathrm{re}}$ if the previous token was not a **Number** or **NegatedMinLong** and a `/` should be interpreted as starting a regular expression, and $NextInputElement^{\mathrm{div}}$ if the previous token was not a **Number** or **NegatedMinLong** 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 num. 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^num depending on
         whether state is re, div, or num, 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 num. 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$^{\mathrm{re}}$ □ *WhiteSpace InputElement*$^{\mathrm{re}}$ (*WhiteSpace*: 7.2)

NextInputElement$^{\mathrm{div}}$ □ *WhiteSpace InputElement*$^{\mathrm{div}}$

NextInputElement$^{\mathrm{num}}$ □ [lookahead□ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace InputElement*$^{\mathrm{div}}$

InputElement$^{\mathrm{re}}$ □
    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$^{\mathrm{div}}$ □
    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*$^{□}$]: INPUTELEMENT;
  Lex[*NextInputElement*$^{\mathrm{re}}$ □ *WhiteSpace InputElement*$^{\mathrm{re}}$] = Lex[*InputElement*$^{\mathrm{re}}$];
  Lex[*NextInputElement*$^{\mathrm{div}}$ □ *WhiteSpace InputElement*$^{\mathrm{div}}$] = Lex[*InputElement*$^{\mathrm{div}}$];
  Lex[*NextInputElement*$^{\mathrm{num}}$ □ [lookahead□ {*ContinuingIdentifierCharacter*, \}] *WhiteSpace InputElement*$^{\mathrm{div}}$]
      = Lex[*InputElement*$^{\mathrm{div}}$];

Lex[*InputElement*$^{□}$]: INPUTELEMENT;
  Lex[*InputElement*$^{□}$ □ *LineBreaks*] = **LineBreak**;
  Lex[*InputElement*$^{□}$ □ *IdentifierOrKeyword*] = Lex[*IdentifierOrKeyword*];
  Lex[*InputElement*$^{□}$ □ *Punctuator*] = Lex[*Punctuator*];
  Lex[*InputElement*$^{\mathrm{div}}$ □ *DivisionPunctuator*] = Lex[*DivisionPunctuator*];
  Lex[*InputElement*$^{□}$ □ *NumericLiteral*] = Lex[*NumericLiteral*];
  Lex[*InputElement*$^{□}$ □ *StringLiteral*] = Lex[*StringLiteral*];
  Lex[*InputElement*$^{\mathrm{re}}$ □ *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.

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 Unicode 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*

Semantics

```
Lex[IdentifierOrKeyword □ IdentifierName]: INPUTELEMENT
   begin
      id: STRING □ LexName[IdentifierName];
      if id □ {“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”, “named”, “namespace”, “native”, “new”,
            “null”, “package”, “private”, “protected”, “public”, “return”, “set”, “static”, “super”,
            “switch”, “synchronized”, “this”, “throw”, “throws”, “transient”, “true”, “try”,
            “typeof”, “use”, “var”, “volatile”, “while”, “with”}
         and IdentifierName contains no escape sequences (i.e. expansions of the NullEscape or HexEscape nonterminals)
      then return the keyword token id
      else return an Identifier token with the name id
      end if
   end;
```

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.

Syntax

IdentifierName □
  InitialIdentifierCharacterOrEscape
  | *NullEscapes InitialIdentifierCharacterOrEscape*
  | *IdentifierName ContinuingIdentifierCharacterOrEscape*
  | *IdentifierName NullEscape*

```
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

```
LexName[IdentifierName]: STRING;
   LexName[IdentifierName → InitialIdentifierCharacterOrEscape] = [LexChar[InitialIdentifierCharacterOrEscape]];
   LexName[IdentifierName → NullEscapes InitialIdentifierCharacterOrEscape]
         = [LexChar[InitialIdentifierCharacterOrEscape]];
   LexName[IdentifierName_0 → IdentifierName_1 ContinuingIdentifierCharacterOrEscape]
         = LexName[IdentifierName_1] ⊕ [LexChar[ContinuingIdentifierCharacterOrEscape]];
   LexName[IdentifierName_0 → IdentifierName_1 NullEscape] = LexName[IdentifierName_1];

LexChar[InitialIdentifierCharacterOrEscape]: CHARACTER;
   LexChar[InitialIdentifierCharacterOrEscape → InitialIdentifierCharacter] = InitialIdentifierCharacter;
   LexChar[InitialIdentifierCharacterOrEscape → \ HexEscape]
      begin
         ch: CHARACTER ← LexChar[HexEscape];
         if ch is in the set of characters accepted by the nonterminal InitialIdentifierCharacter then return ch
         else throw syntaxError
         end if
      end;

LexChar[ContinuingIdentifierCharacterOrEscape]: CHARACTER;
   LexChar[ContinuingIdentifierCharacterOrEscape → ContinuingIdentifierCharacter]
         = ContinuingIdentifierCharacter;
   LexChar[ContinuingIdentifierCharacterOrEscape → \ HexEscape]
      begin
         ch: CHARACTER ← LexChar[HexEscape];
         if ch is in the set of characters accepted by the nonterminal ContinuingIdentifierCharacter then return ch
         else throw syntaxError
         end if
      end;
```

The characters in the specified categories in version 3.0 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*]: TOKEN = the punctuator token *Punctuator*.

Lex[*DivisionPunctuator*]: TOKEN = the punctuator token *DivisionPunctuator*.

7.7 Numeric literals

Syntax

```
NumericLiteral □
    DecimalLiteral
  | HexIntegerLiteral
  | DecimalLiteral LetterF
  | IntegerLiteral LetterL
  | IntegerLiteral LetterU LetterL
```

```
IntegerLiteral □
    DecimalIntegerLiteral
  | HexIntegerLiteral
```

```
LetterF □ F | f
```

```
LetterL □ L | l
```

LetterU □ `U` | `u`

DecimalLiteral □
    Mantissa
  | *Mantissa LetterE SignedInteger*

LetterE □ `E` | `e`

Mantissa □
    DecimalIntegerLiteral
  | *DecimalIntegerLiteral* `.`
  | *DecimalIntegerLiteral* `.` *Fraction*
  | `.` *Fraction*

DecimalIntegerLiteral □
    `0`
  | *NonZeroDecimalDigits*

NonZeroDecimalDigits □
    NonZeroDigit
  | *NonZeroDecimalDigits ASCIIDigit*

Fraction □ *DecimalDigits*

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]: TOKEN;
   Lex[NumericLiteral □ DecimalLiteral] = a Number token with the value
         realToFloat64(LexNumber[DecimalLiteral]);
   Lex[NumericLiteral □ HexIntegerLiteral] = a Number token with the value
         realToFloat64(LexNumber[HexIntegerLiteral]);
   Lex[NumericLiteral □ DecimalLiteral LetterF] = a Number token with the value
         realToFloat32(LexNumber[DecimalLiteral]);
   Lex[NumericLiteral □ IntegerLiteral LetterL]
      begin
         i: INTEGER □ LexNumber[IntegerLiteral];
         if $i \le 2^{63} - 1$ then return a Number token with the value LONG□value: i□
         elsif $i = 2^{63}$ then return NegatedMinLong
         else throw rangeError
         end if
      end;
   Lex[NumericLiteral □ IntegerLiteral LetterU LetterL]
      begin
         i: INTEGER □ LexNumber[IntegerLiteral];
         if $i \le 2^{64} - 1$ then return a Number token with the value ULONG□value: i□else throw rangeError end if
      end;

LexNumber[IntegerLiteral]: INTEGER;
   LexNumber[IntegerLiteral □ DecimalIntegerLiteral] = LexNumber[DecimalIntegerLiteral];
   LexNumber[IntegerLiteral □ HexIntegerLiteral] = LexNumber[HexIntegerLiteral];
```

NOTE Note that all digits of hexadecimal literals are significant.

```
LexNumber[DecimalLiteral]: RATIONAL;
   LexNumber[DecimalLiteral □ Mantissa] = LexNumber[Mantissa];
   LexNumber[DecimalLiteral □ Mantissa LetterE SignedInteger] = LexNumber[Mantissa]□$10^{\text{LexNumber[SignedInteger]}}$;

LexNumber[Mantissa]: RATIONAL;
   LexNumber[Mantissa □ DecimalIntegerLiteral] = LexNumber[DecimalIntegerLiteral];
   LexNumber[Mantissa □ DecimalIntegerLiteral .] = LexNumber[DecimalIntegerLiteral];
   LexNumber[Mantissa □ DecimalIntegerLiteral . Fraction]
         = LexNumber[DecimalIntegerLiteral] + LexNumber[Fraction];
   LexNumber[Mantissa □ . Fraction] = LexNumber[Fraction];

LexNumber[DecimalIntegerLiteral]: INTEGER;
   LexNumber[DecimalIntegerLiteral □ 0] = 0;
   LexNumber[DecimalIntegerLiteral □ NonZeroDecimalDigits] = LexNumber[NonZeroDecimalDigits];

LexNumber[NonZeroDecimalDigits]: INTEGER;
   LexNumber[NonZeroDecimalDigits □ NonZeroDigit] = DecimalValue[NonZeroDigit];
   LexNumber[$NonZeroDecimalDigits_0$ □ $NonZeroDecimalDigits_1$ ASCIIDigit]
         = 10□LexNumber[$NonZeroDecimalDigits_1$] + DecimalValue[ASCIIDigit];

LexNumber[Fraction □ DecimalDigits]: RATIONAL = LexNumber[DecimalDigits]/$10^{\text{NDigits[DecimalDigits]}}$;

LexNumber[SignedInteger]: INTEGER;
   LexNumber[SignedInteger □ DecimalDigits] = LexNumber[DecimalDigits];
   LexNumber[SignedInteger □ + DecimalDigits] = LexNumber[DecimalDigits];
   LexNumber[SignedInteger □ - DecimalDigits] = –LexNumber[DecimalDigits];
```

LexNumber[*DecimalDigits*]: INTEGER;
LexNumber[*DecimalDigits* □ *ASCIIDigit*] = DecimalValue[*ASCIIDigit*];
LexNumber[*DecimalDigits*$_0$ □ *DecimalDigits*$_1$ *ASCIIDigit*]
= 10□LexNumber[*DecimalDigits*$_1$] + DecimalValue[*ASCIIDigit*];

NDigits[*DecimalDigits*]: INTEGER;
NDigits[*DecimalDigits* □ *ASCIIDigit*] = 1;
NDigits[*DecimalDigits*$_0$ □ *DecimalDigits*$_1$ *ASCIIDigit*] = NDigits[*DecimalDigits*$_1$] + 1;

LexNumber[*HexIntegerLiteral*]: INTEGER;
LexNumber[*HexIntegerLiteral* □ `0` *LetterX* *HexDigit*] = HexValue[*HexDigit*];
LexNumber[*HexIntegerLiteral*$_0$ □ *HexIntegerLiteral*$_1$ *HexDigit*]
= 16□LexNumber[*HexIntegerLiteral*$_1$] + HexValue[*HexDigit*];

DecimalValue[*ASCIIDigit*]: INTEGER = *ASCIIDigit*’s decimal value (an integer between 0 and 9).

DecimalValue[*NonZeroDigit*] = *NonZeroDigit*’s decimal value (an integer between 1 and 9).

HexValue[*HexDigit*]: INTEGER = *HexDigit*’s hexadecimal 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*$^{\text{single}}$ `'`
| `"` *StringChars*$^{\text{double}}$ `"`

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

StringChar$^{□}$ □
LiteralStringChar$^{□}$
| `\` *StringEscape*

LiteralStringChar$^{\text{single}}$ □ *UnicodeCharacter* **except** `'` | `\` | *LineTerminator* (*UnicodeCharacter*: 7.3)

LiteralStringChar$^{\text{double}}$ □ *UnicodeCharacter* **except** `"` | `\` | *LineTerminator* (*LineTerminator*: 7.3)

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*]: Token;
  Lex[*StringLiteral* □ `'` *StringChars*$^{\text{single}}$ `'`] = a **String** token with the value LexString[*StringChars*$^{\text{single}}$];
  Lex[*StringLiteral* □ `"` *StringChars*$^{\text{double}}$ `"`] = a **String** token with the value LexString[*StringChars*$^{\text{double}}$];

LexString[*StringChars*$^{□}$]: String;
  LexString[*StringChars*$^{□}$ □ «empty»] = “”;
  LexString[*StringChars*$^{□}_{0}$ □ *StringChars*$^{□}_{1}$ *StringChar*$^{□}$] = LexString[*StringChars*$^{□}_{1}$] ⊕ [LexChar[*StringChar*$^{□}$]];
  LexString[*StringChars*$^{□}_{0}$ □ *StringChars*$^{□}_{1}$ *NullEscape*] = LexString[*StringChars*$^{□}_{1}$];

LexChar[*StringChar*$^{□}$]: Character;
  LexChar[*StringChar*$^{□}$ □ *LiteralStringChar*$^{□}$] = *LiteralStringChar*$^{□}$;
  LexChar[*StringChar*$^{□}$ □ `\` *StringEscape*] = LexChar[*StringEscape*];

LexChar[*StringEscape*]: Character;
  LexChar[*StringEscape* □ *ControlEscape*] = LexChar[*ControlEscape*];
  LexChar[*StringEscape* □ *ZeroEscape*] = LexChar[*ZeroEscape*];
  LexChar[*StringEscape* □ *HexEscape*] = LexChar[*HexEscape*];
  LexChar[*StringEscape* □ *IdentityEscape*] = *IdentityEscape*;

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

LexChar[*ControlEscape*]: Character;
  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*}]]: Character = ‘«NUL»’;

LexChar[*HexEscape*]: Character;
  LexChar[*HexEscape* □ `x` *HexDigit*$_1$ *HexDigit*$_2$]
    = *codeToCharacter*(16□HexValue[*HexDigit*$_1$] + HexValue[*HexDigit*$_2$]);
  LexChar[*HexEscape* □ `u` *HexDigit*$_1$ *HexDigit*$_2$ *HexDigit*$_3$ *HexDigit*$_4$]
    = *codeToCharacter*(4096□HexValue[*HexDigit*$_1$] + 256□HexValue[*HexDigit*$_2$] + 16□HexValue[*HexDigit*$_3$] + HexValue[*HexDigit*$_4$]);

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»
| *RegExpFlags ContinuingIdentifierCharacterOrEscape* (*ContinuingIdentifierCharacterOrEscape*: 7.5)
| *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*]: TOKEN
  = A **RegularExpression** token with the body LexString[*RegExpBody*] and flags LexString[*RegExpFlags*];

LexString[*RegExpFlags*]: STRING;
  LexString[*RegExpFlags* □ «empty»] = “”;
  LexString[*RegExpFlags*$_0$ □ *RegExpFlags*$_1$ *ContinuingIdentifierCharacterOrEscape*]
    = LexString[*RegExpFlags*$_1$] ⊕ [LexChar[*ContinuingIdentifierCharacterOrEscape*]];
  LexString[*RegExpFlags*$_0$ □ *RegExpFlags*$_1$ *NullEscape*] = LexString[*RegExpFlags*$_1$];

LexString[*RegExpBody* □ `/` [lookahead□{`*`}] *RegExpChars* `/`]: STRING = LexString[*RegExpChars*];

LexString[*RegExpChars*]: STRING;
  LexString[*RegExpChars* □ *RegExpChar*] = LexString[*RegExpChar*];
  LexString[*RegExpChars*$_0$ □ *RegExpChars*$_1$ *RegExpChar*] = LexString[*RegExpChars*$_1$] ⊕ LexString[*RegExpChar*];

LexString[*RegExpChar*]: STRING;
  LexString[*RegExpChar* □ *OrdinaryRegExpChar*] = [*OrdinaryRegExpChar*];
  LexString[*RegExpChar* □ `\` *NonTerminator*] = [‘`\`’, *NonTerminator*]; (Note that the result string has two characters)

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 signed or unsigned 64-bit integer, a single or double-precision floating-point number, a character, a string, a namespace, a compound attribute, a class, a simple instance, a method closure, a date, a regular expression, 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 ∪ Long ∪ ULong ∪ Float32 ∪ Float64 ∪ Character ∪ String ∪ Namespace ∪ CompoundAttribute ∪ Class ∪ SimpleInstance ∪ MethodClosure ∪ Date ∪ RegExp ∪ Package ∪ GlobalObject;

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

PrimitiveObject
= Undefined ∪ Null ∪ Boolean ∪ Long ∪ ULong ∪ Float32 ∪ Float64 ∪ Character ∪ String;

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**};

The semantic domain IntegerOpt consists of all integers as well as **none**:

IntegerOpt = Integer ∪ {**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 Booleans

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

9.1.4 Numbers

The semantic domains Long, ULong, Float32, and Float64, collectively denoted by the domain GeneralNumber, represent the numeric types supported by ECMAScript. See section 5.12.

9.1.5 Strings

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

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 Namespaces

A namespace object is represented by a Namespace record (see section 5.11) 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 Qualified Names

A QualifiedName tuple (see section 5.10) has the fields below and represents a name qualified with a namespace.

Field	Contents	Note
namespace	Namespace	The namespace qualifier
id	String	The name

MultiName is the semantic domain of sets of qualified names. Multinames are used internally in property lookup.

MultiName = QualifiedName{}

9.1.7 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 5.10) 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**, **abstract**, **virtual**, or **final** if one of these attributes has been given; **none** if not. MEMBERMODIFIER = {**none**, **static**, **constructor**, **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 Classes

Programmer-visible class objects are represented as CLASS records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Map of qualified names to static members defined in this class (see section *****)
parent	CLASSOPT	This class's immediate superclass or **null** if none
instanceBindings	INSTANCEBINDING{}	Map of qualified names to vtable indices of instance members defined in this class
vTable	VTABLEENTRY{}	Map of vtable indices to 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**
typeofString	STRING	A string to return if `typeof` is invoked on this class's instances
privateNamespace	NAMESPACE	This class's `private` namespace
dynamic	BOOLEAN	**true** if this class or any of its ancestors was defined with the `dynamic` attribute
final	BOOLEAN	**true** if this class cannot be subclassed
call	OBJECT □ ARGUMENTLIST □ PHASE □ OBJECT	A procedure to call when this class is used in a call expression
construct	ARGUMENTLIST □ PHASE □ OBJECT	A procedure to call when this class is used in a `new` expression

		expression
isInstanceOf	OBJECT □ BOOLEAN	A procedure to call to determine whether a given object is an instance of this class
implicitCoerce	OBJECT □ BOOLEAN □ OBJECT	A procedure to call when a value is assigned to a variable, parameter, or result whose type is this class. The argument to implicitCoerce can be any value, which may or may not be an instance of this class; the result must be an instance of this class. If the coercion is not appropriate, implicitCoerce should throw an exception if its second argument is **false** or return **null** (as long as **null** is an instance of this class) if its second argument is **true**.
defaultValue	OBJECT	When a variable whose type is this class is defined but not explicitly initialised, the variable's initial value is defaultValue, which must be an instance of this class.

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 Simple Instances

Instances of programmer-defined classes as well as of some built-in classes are represented as SIMPLEINSTANCE records (see section 5.11) with the fields below. Prototype-based objects are also SIMPLEINSTANCE records.

Field	**Contents**	**Note**
localBindings	LOCALBINDING{}	Map of qualified names to local properties (including dynamic properties, if any) of this instance
parent	OBJECTOPT	This instance's parent link
sealed	BOOLEAN	If **true**, no more local properties may be added to this instance
type	CLASS	This instance's type
slots	SLOT{}	A set of slots that hold this instance's fixed property values
call	OBJECT □ ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT □ {**none**}	Either **none** or 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 □ {**none**}	Either **none** or 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
toClass	CLASSOPT	Either **none** or a class to use if this instance is used in a place where a class is expected
env	ENVIRONMENTOPT	Either **none** or the environment to pass to the call or construct procedure

9.1.9.1 Slots

A SLOT record (see section 5.11) 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

9.1.10 Uninstantiated Functions

An UNINSTANTIATEDFUNCTION record (see section 5.11) has the fields below. It is not an instance in itself but creates a SIMPLEINSTANCE when instantiated with an environment. UNINSTANTIATEDFUNCTION records represent functions with variables inherited from their enclosing environments; supplying the environment turns such a function into a SIMPLEINSTANCE.

Field	Contents	Note
type	CLASS	Values to be transferred into the generated SIMPLEINSTANCE's corresponding fields
defaultSlots	SLOT{}	A list of the default values of the generated SIMPLEINSTANCE's slots
buildPrototype	BOOLEAN	If true, the generated SIMPLEINSTANCE gets a separate `prototype` slot with its own protype object
call	OBJECT □ ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT □ {**none**}	Values to be transferred into the generated SIMPLEINSTANCE's corresponding fields
construct	ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT □ {**none**}	
instantiations	SIMPLEINSTANCE{}	Set of prior instantiations. This set serves only to precisely specify the closure sharing optimization and would not be needed in any actual implementation.

9.1.11 Method Closures

A METHODCLOSURE tuple (see section 5.10) 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

9.1.12 Dates

Instances of the `Date` class are represented as DATE records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Same as in SIMPLEINSTANCEs (section 9.1.9)
parent	OBJECTOPT	
sealed	BOOLEAN	
timeValue	INTEGER	The date expressed as a count of milliseconds from January 1, 1970 UTC

9.1.13 Regular Expressions

Instances of the `RegExp` class are represented as REGEXP records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Same as in SIMPLEINSTANCEs (section 9.1.9)
parent	OBJECTOPT	
sealed	BOOLEAN	
source	STRING	This regular expression's source pattern
lastIndex	INTEGER	The string position at which to start the next regular expression match
global	BOOLEAN	**true** if the regular expression flags included the flag `g`
ignoreCase	BOOLEAN	**true** if the regular expression flags included the flag `i`
multiline	BOOLEAN	**true** if the regular expression flags included the flag `m`

9.1.14 Packages

Programmer-visible packages are represented as PACKAGE records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Map of qualified names to members defined in this package
internalNamespace	NAMESPACE	This package's `internal` namespace

9.1.15 Global Objects

Programmer-visible global objects are represented as GLOBALOBJECT records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Map of qualified names to members defined in this global object
parent	OBJECTOPT	This global object's parent link
sealed	BOOLEAN	If **true**, no more properties may be added to this global object
internalNamespace	NAMESPACE	This global object's `internal` namespace

9.2 Objects with Limits

A LIMITEDINSTANCE tuple (see section 5.10) represents an intermediate result of a `super` or `super(`*expr*`)` subexpression. It has the fields below.

Field	Contents	Note
instance	OBJECT	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 5.10) 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	Boolean	**true** if strict mode was in effect at the point where the reference was created

A DotReference tuple (see section 5.10) 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 5.10) 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)

9.4 Function Support

There are three kinds of functions: normal functions, getters, and setters. The FunctionKind semantic domain encodes the kind:

FunctionKind = {**normal**, **get**, **set**}

9.5 Argument Lists

An ArgumentList tuple (see section 5.10) 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 5.10) 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

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**}

9.7 Contexts

A CONTEXT tuple (see section 5.10) 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.

9.8 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 5.10) 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

9.9 Environment Frames

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 GLOBALOBJECT.

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**};

The semantic domain EnvironmentOpt consists of all environments as well as the tag **none** which denotes the absence of an environment:

EnvironmentOpt = Environment □ {**none**};

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 □ GlobalObject □ Package □ ParameterFrame □ 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**}

9.9.1 System Frame

The top-level frame containing predefined constants, functions, and classes is represented as a SystemFrame record (see section 5.11) with the field below.

Field	Contents	Note
localBindings	LocalBinding{}	Map of qualified names to definitions in this frame

9.9.2 Function Parameter Frames

Frames holding bindings for invoked functions are represented as ParameterFrame records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LocalBinding{}	Map of qualified names to definitions in this function
plurality	Plurality	See section 9.9
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
unchecked	Boolean	**true** if this function's arguments are not checked against its parameter signature
prototype	Boolean	**true** if this function is not an instance method but defines `this` anyway
positional	Parameter[]	List of this function's positional parameters
named	NamedParameter{}	Set of this function's named parameters
rest	Variable □ {**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	**true** if this function is not an instance method but defines `this` anyway

9.9.2.1 Parameters

A PARAMETER tuple (see section 5.10) has the fields below and represents the signature of one positional parameter.

Field	Contents	Note
var	DYNAMICVAR □ VARIABLE	The local variable that will hold this parameter's value
default	OBJECTOPT	This parameter's default value; if **none**, this parameter is required

A NAMEDPARAMETER tuple (see section 5.10) has the fields below and represents the signature of one named parameter.

Field	Contents	Note
name	STRING	This parameter's external name
var	VARIABLE	The local variable that will hold this parameter's value
default	OBJECT	This parameter's default value

9.9.3 Block Frames

Frames holding bindings for blocks are represented as BLOCKFRAME records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LOCALBINDING{}	Map of qualified names to definitions in this block
plurality	PLURALITY	See section 9.9

9.10 Environment Bindings

In general, accesses of members are either read or write operations. The tags **read** and **write** indicate these respectively. The semantic domain ACCESS consists of these two tags:

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

Some members are visible only for read or only for write accesses; other members are visible to both read and write accesses. The tag **readWrite** indicates that a member is visible to both kinds of accesses. The semantic domain ACCESSSET consists of the three possible access visibilities:

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

NOTE Access sets indicate visibility, not permission to perform the desired access. Immutable members generally have the access **readWrite** but an attempt to write one results in an error. Trying to write to member with the access **read** would not even find the member, and the write would proceed to search an object's parent hierarchy for another matching member.

9.10.1 Static Bindings

A LOCALBINDING tuple (see section 5.10) 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
accesses	ACCESSSET	Accesses for which this member is visible
content	LOCALMEMBER	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 local member is either **forbidden**, a dynamic variable, a variable, a constructor method, a getter, or a setter:

LOCALMEMBER = {**forbidden**} □ DYNAMICVAR □ VARIABLE □ CONSTRUCTORMETHOD □ GETTER □ SETTER;

LOCALMEMBEROPT = LOCALMEMBER □ {**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 5.11) 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 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

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 uninstantiated function (compile time only), or a semantic procedure (compile time only) that takes no parameters and will compute an object on demand; 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**} □ UNINSTANTIATEDFUNCTION □ () □ OBJECT;

A DYNAMICVAR record (see section 5.11) has the fields below and describes one hoisted or dynamic variable.

Field	Contents	Note
value	OBJECT □ UNINSTANTIATEDFUNCTION	This variable's current value; may be an uninstantiated function at compile time
sealed	BOOLEAN	**true** if this variable cannot be deleted using the `delete` operator

A CONSTRUCTORMETHOD record (see section 5.11) has the field below and describes one constructor definition.

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

A GETTER record (see section 5.11) 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 5.11) 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

9.10.2 Instance Bindings

An INSTANCEBINDING tuple (see section 5.10) has the fields below and describes the binding of one qualified name to a vtable index. 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 and the other binding is for writing).

Field	**Contents**	**Note**
qname	QUALIFIEDNAME	The qualified name bound by this binding
access	ACCESS	Access for which this member is visible
index	VTABLEINDEX	The vtable index to which this qualified name was bound

A VTABLEINDEX record (see section 5.11) has no fields and is merely used as a unique value.

VTABLEINDEXOPT = VTABLEINDEX □ {**none**};

A VTABLEENTRY tuple (see section 5.10) has the fields below and describes the binding of a vtable index to an instance member.

Field	**Contents**	**Note**
index	VTABLEINDEX	The vtable index name bound by this entry
content	INSTANCEMEMBER	The instance member to which this vtable index was bound

An instance member is either an instance variable, an instance method, or an instance accessor:
INSTANCEMEMBER = INSTANCEVARIABLE □ INSTANCEMETHOD □ INSTANCEGETTER □ INSTANCESETTER;

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

An INSTANCEVARIABLE record (see section 5.11) has the fields below and describes one instance variable or constant definition.

Field	**Contents**	**Note**
final	BOOLEAN	**true** if this member may not be overridden in subclasses
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

An INSTANCEMETHOD record (see section 5.11) has the fields below and describes one instance method definition.

Field	**Contents**	**Note**
final	BOOLEAN	**true** if this member may not be overridden in subclasses
signature	PARAMETERFRAME	This method's signature
call	OBJECT □ ARGUMENTLIST □ PHASE □ OBJECT	A procedure to call when this instance method is invoked. The procedure takes a `this` OBJECT, an ARGUMENTLIST (see section 9.5), and a PHASE (see section 9.6) and produces an OBJECT result

An INSTANCEGETTER record (see section 5.11) has the fields below and describes one instance getter definition.

Field	Contents	Note
final	BOOLEAN	**true** if this member may not be overridden in subclasses
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

An INSTANCESETTER record (see section 5.11) has the fields below and describes one instance setter definition.

Field	Contents	Note
final	BOOLEAN	**true** if this member may not be overridden in subclasses
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

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 $j \geq 2^{31}$ 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 $j \ge 2^{63}$ then j ← $j – 2^{64}$ end if;
   return j
end proc;

proc truncateToInteger(x: GENERALNUMBER): INTEGER
   case x of
      {$+\infty_{f32}$, $+\infty_{f64}$, $–\infty_{f32}$, $–\infty_{f64}$, $NaN32_{f32}$, $NaN64_{f64}$} do return 0;
      FINITEFLOAT32 do return truncateFiniteFloat32(x);
      FINITEFLOAT64 do return truncateFiniteFloat64(x);
      LONG ∪ ULONG do return x.value
   end case
end proc;

proc checkInteger(x: GENERALNUMBER): INTEGEROPT
   case x of
      {$NaN32_{f32}$, $NaN64_{f64}$, $+\infty_{f32}$, $+\infty_{f64}$, $–\infty_{f32}$, $–\infty_{f64}$} do return none;
      {$+zero_{f32}$, $+zero_{f64}$, $–zero_{f32}$, $–zero_{f64}$} do return 0;
      LONG ∪ ULONG do return x.value;
      NONZEROFINITEFLOAT32 ∪ NONZEROFINITEFLOAT64 do
         r: RATIONAL ← x.value;
         if r ∉ INTEGER then return none end if;
         return r
   end case
end proc;

proc integerToLong(i: INTEGER): GENERALNUMBER
   if $–2^{63} \le i \le 2^{63} – 1$ then return LONG〈value: i〉
   elsif $2^{63} \le i \le 2^{64} – 1$ then return ULONG〈value: i〉
   else return realToFloat64(i)
   end if
end proc;

proc integerToULong(i: INTEGER): GENERALNUMBER
   if $0 \le i \le 2^{64} – 1$ then return ULONG〈value: i〉
   elsif $–2^{63} \le i \le –1$ then return LONG〈value: i〉
   else return realToFloat64(i)
   end if
end proc;

proc rationalToLong(q: RATIONAL): GENERALNUMBER
   if q ∈ INTEGER then return integerToLong(q)
   elsif $|q| \le 2^{53}$ then return realToFloat64(q)
   elsif $q < –2^{63} – 1/2$ or $q \ge 2^{64} – 1/2$ then return realToFloat64(q)
   else
      Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
      note $–2^{63} \le i \le 2^{64} – 1$;
      if $i < 2^{63}$ then return LONG〈value: i〉 else return ULONG〈value: i〉 end if
   end if
end proc;
```

```
proc rationalToULong(q: RATIONAL): GENERALNUMBER
    if q ∈ INTEGER then return integerToULong(q)
    elsif $|q| \le 2^{53}$ then return realToFloat64(q)
    elsif $q < –2^{63} – 1/2$ or $q \ge 2^{64} – 1/2$ then return realToFloat64(q)
    else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        note $–2^{63} \le i \le 2^{64} – 1$;
        if $i \ge 0$ then return ULONG[value: i] else return LONG[value: i] end if
    end if
end proc;
```

```
proc toRational(x: FINITEGENERALNUMBER): RATIONAL
    case x of
        {+zero_f32, +zero_f64, –zero_f32, –zero_f64} do return 0;
        NONZEROFINITEFLOAT32 ∪ NONZEROFINITEFLOAT64 ∪ LONG ∪ ULONG do return x.value
    end case
end proc;
```

```
proc toFloat64(x: GENERALNUMBER): FLOAT64
    case x of
        LONG ∪ ULONG do return realToFloat64(x.value);
        FLOAT32 do return float32ToFloat64(x);
        FLOAT64 do return x
    end case
end proc;
```

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

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

```
proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
    if x ∈ {NaN32_f32, NaN64_f64} or y ∈ {NaN32_f32, NaN64_f64} then return unordered
    elsif x ∈ {+∞_f32, +∞_f64} and y ∈ {+∞_f32, +∞_f64} then return equal
    elsif x ∈ {–∞_f32, –∞_f64} and y ∈ {–∞_f32, –∞_f64} then return equal
    elsif x ∈ {+∞_f32, +∞_f64} or y ∈ {–∞_f32, –∞_f64} then return greater
    elsif x ∈ {–∞_f32, –∞_f64} or y ∈ {+∞_f32, +∞_f64} then return less
    else
        xr: RATIONAL ← toRational(x);
        yr: RATIONAL ← toRational(y);
        if xr < yr then return less
        elsif xr > yr then return greater
        else return equal
        end if
    end if
end proc;
```

10.2 Object Utilities

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;
        LONG do return longClass;
        ULONG do return uLongClass;
        FLOAT32 do return floatClass;
        FLOAT64 do return numberClass;
        CHARACTER do return characterClass;
        STRING do return stringClass;
        NAMESPACE do return namespaceClass;
        COMPOUNDATTRIBUTE do return attributeClass;
        CLASS do return classClass;
        SIMPLEINSTANCE do return o.type;
        METHODCLOSURE do return functionClass;
        DATE do return dateClass;
        REGEXP do return regExpClass;
        PACKAGE □ GLOBALOBJECT do return packageClass
    end case
end proc;
```

10.2.2 *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;
        LONG □ ULONG do return o.value $\neq$ 0;
        FLOAT32 do return o □ {+zero$_{f32}$, –zero$_{f32}$, NaN32$_{f32}$};
        FLOAT64 do return o □ {+zero$_{f64}$, –zero$_{f64}$, NaN64$_{f64}$};
        STRING do return o $\neq$ “”;
        CHARACTER □ NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ SIMPLEINSTANCE □ METHODCLOSURE □
                DATE □ REGEXP □ PACKAGE □ GLOBALOBJECT do
            return true
    end case
end proc;
```

10.2.3 *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 NaN64$_{f64}$;
        NULL □ {false} do return +zero$_{f64}$;
        {true} do return 1.0$_{f64}$;
        GENERALNUMBER do return o;
        CHARACTER □ STRING do ????;
        NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ METHODCLOSURE □ PACKAGE □ GLOBALOBJECT do
            throw badValueError;
        SIMPLEINSTANCE do ????;
        DATE do ????;
        REGEXP do ????
    end case
end proc;
```

10.2.4 *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”;
        LONG □ ULONG do return integerToString(o.value);
        FLOAT32 do return float32ToString(o);
        FLOAT64 do return float64ToString(o);
        CHARACTER do return [o];
        STRING do return o;
        NAMESPACE do ????;
        COMPOUNDATTRIBUTE do ????;
        CLASS do ????;
        METHODCLOSURE do ????;
        SIMPLEINSTANCE do ????;
        DATE do ????;
        REGEXP do ????;
        PACKAGE □ GLOBALOBJECT 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;
```

integerToStringWithSign(*i*) is the same as *integerToString*(*i*) except that the resulting string always begins with a plus or minus sign.

```
proc integerToStringWithSign(i: INTEGER): STRING
    if $i \geq 0$ then return [‘+’] ⊕ integerToString(i)
    else return [‘-’] ⊕ integerToString(–i)
    end if
end proc;
```

float32ToString(*x*) converts a FLOAT32 *x* to a string using fixed-point notation if the absolute value of *x* is between 10^{-6} inclusive and 10^{21} exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a FLOAT32 value would result in the same value *x* (except that **–zero**$_{f32}$ would become **+zero**$_{f32}$).

```
proc float32ToString(x: FLOAT32): STRING
  case x of
    {NaN32_f32} do return “NaN”;
    {+zero_f32, –zero_f32} do return “0”;
    {+∞_f32} do return “Infinity”;
    {–∞_f32} do return “-Infinity”;
    NONZEROFINITEFLOAT32 do
      r: RATIONAL □ x.value;
      if $r < 0$ then return “-” ⊕ float32ToString(float32Negate(x))
      else
        Let n, k, and s be integers such that $k \geq 1$, $10^{k-1} \leq s \leq 10^k$, realToFloat32(s□$10^{n-k}$) = x, and k is as small as
            possible.
        When there are multiple possibilities for s according to the rules above, implementations are encouraged but
            not required to select the one according to the following rules: Select the value of s for which s□$10^{n-k}$ is
            closest in value to r; if there are two such possible values of s, choose the one that is even.
        digits: STRING □ integerToString(s);
        if $k \leq n \leq 21$ then return digits ⊕ repeat(‘0’, $n - k$)
        elsif $0 < n \leq 21$ then return digits[0 ... $n - 1$] ⊕ “.” ⊕ digits[n ...]
        elsif $–6 < n \leq 0$ then return “0.” ⊕ repeat(‘0’, $–n$) ⊕ digits
        else
          mantissa: STRING;
          if $k = 1$ then mantissa □ digits
          else mantissa □ digits[0 ... 0] ⊕ “.” ⊕ digits[1 ...]
          end if;
          return mantissa ⊕ “e” ⊕ integerToStringWithSign($n – 1$)
        end if
      end if
  end case
end proc;
```

float64ToString(*x*) converts a FLOAT64 *x* to a string using fixed-point notation if the absolute value of *x* is between 10^{-6} inclusive and 10^{21} exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a FLOAT64 value would result in the same value *x* (except that **–zero**$_{f64}$ would become **+zero**$_{f64}$).

```
proc float64ToString(x: FLOAT64): STRING
    case x of
        {NaN64_f64} do return "NaN";
        {+zero_f64, –zero_f64} do return "0";
        {+∞_f64} do return "Infinity";
        {–∞_f64} do return "-Infinity";
        NONZEROFINITEFLOAT64 do
            r: RATIONAL ← x.value;
            if $r < 0$ then return "-" ⊕ float64ToString(float64Negate(x))
            else
                Let n, k, and s be integers such that $k \geq 1$, $10^{k-1} \leq s \leq 10^{k}$, $realToFloat64(s \times 10^{n-k}) = x$, and k is as small as
                        possible.
                When there are multiple possibilities for s according to the rules above, implementations are encouraged but
                        not required to select the one according to the following rules: Select the value of s for which $s \times 10^{n-k}$ is
                        closest in value to r; if there are two such possible values of s, choose the one that is even.
                digits: STRING ← integerToString(s);
                if $k \leq n \leq 21$ then return digits ⊕ repeat('0', $n - k$)
                elsif $0 < n \leq 21$ then return digits[0 ... $n - 1$] ⊕ "." ⊕ digits[n ...]
                elsif $–6 < n \leq 0$ then return "0." ⊕ repeat('0', $–n$) ⊕ digits
                else
                    mantissa: STRING;
                    if $k = 1$ then mantissa ← digits
                    else mantissa ← digits[0 ... 0] ⊕ "." ⊕ digits[1 ...]
                    end if;
                    return mantissa ⊕ "e" ⊕ integerToStringWithSign($n – 1$)
                end if
            end if
    end case
end proc;
```

10.2.5 *toPrimitive*

```
proc toPrimitive(o: OBJECT, hint: OBJECT, phase: PHASE): PRIMITIVEOBJECT
    case o of
        PRIMITIVEOBJECT do return o;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ SIMPLEINSTANCE ∪ METHODCLOSURE ∪ REGEXP ∪
                PACKAGE ∪ GLOBALOBJECT do
            return toString(o, phase);
        DATE do ????
    end case
end proc;
```

10.2.6 *toClass*

```
proc toClass(o: OBJECT): CLASS
    case o of
        CLASS do return o;
        SIMPLEINSTANCE do
            c: CLASSOPT ← o.toClass;
            if $c \neq$ none then return c else throw badValueError end if;
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHARACTER ∪ STRING ∪
                NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ METHODCLOSURE ∪ DATE ∪ REGEXP ∪ PACKAGE ∪
                GLOBALOBJECT do
            throw badValueError
    end case
end proc;
```

10.2.7 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
      note At this point 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
   note o must be a SIMPLEINSTANCE.
   matchingSlots: SLOT{} ← {s | ∀s ∈ 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 ∪ GLOBALOBJECT
   g: FRAME ← env[|env| – 2];
   note The penultimate frame g is always a PACKAGE or GLOBALOBJECT.
   return g
end proc;
```

10.5.1 Access Utilities

```
proc accessesOverlap(accesses1: ACCESSSET, accesses2: ACCESSSET): BOOLEAN
   return accesses1 = accesses2 or accesses1 = readWrite or accesses2 = readWrite
end proc;
```

10.5.2 Adding Local Definitions

```
proc defineLocalMember(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE{},
        overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, accesses: ACCESSSET, m: LOCALMEMBER): 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);
    if some b □ localFrame.localBindings satisfies
            b.qname □ multiname and accessesOverlap(b.accesses, accesses) then
        throw definitionError
    end if;
    for each frame □ regionalEnv[1 ...] do
        if some b □ frame.localBindings satisfies b.qname □ multiname and accessesOverlap(b.accesses, accesses) and
                b.content ≠ forbidden then
            throw definitionError
        end if
    end for each;
    newBindings: LOCALBINDING{} □
            {LOCALBINDING□qname: qname, accesses: accesses, content: m, explicit: explicit□| □qname □ multiname};
    localFrame.localBindings □ localFrame.localBindings □ newBindings;
    note 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: LOCALBINDING{} □
            {LOCALBINDING□qname: qname, accesses: accesses, content: forbidden, explicit: true□|
            □qname □ multiname};
    for each frame □ regionalEnv[1 ...] do
        frame.localBindings □ frame.localBindings □ newForbiddenBindings
    end for each;
    return multiname
end proc;
```

defineHoistedVar(*env*, *id*, *initialValue*) defines a hoisted variable with the name *id* in the environment *env*. Hoisted variables are hoisted to the global or enclosing function scope. Multiple hoisted variables may be defined in the same scope, but they may not coexist with non-hoisted variables with the same name. A hoisted variable can be defined using either a `var` or a `function` statement. If it is defined using `var`, then *initialValue* is always **undefined** (if the `var` statement has an initialiser, then the variable's value will be written later when the `var` statement is executed). If it is defined using `function`, then *initialValue* must be a function instance or open instance. A `var` hoisted variable may be hoisted into the PARAMETERFRAME if there is already a parameter with the same name; a `function` hoisted variable is never hoisted into the PARAMETERFRAME and will shadow a parameter with the same name for compatibility with ECMAScript Edition 3. If there are multiple `function` definitions, the initial value is the last `function` definition.

```
proc defineHoistedVar(env: ENVIRONMENT, id: STRING, initialValue: OBJECT □ UNINSTANTIATEDFUNCTION):
      DYNAMICVAR
   qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: publicNamespace, id: id□;
   regionalEnv: FRAME[] □ getRegionalEnvironment(env);
   regionalFrame: FRAME □ regionalEnv[|regionalEnv| – 1];
   note env is either the GLOBALOBJECT or a PARAMETERFRAME because hoisting only occurs into global or function
         scope.
   existingBindings: LOCALBINDING{} □ {b | □b □ regionalFrame.localBindings such that b.qname = qname};
   if (existingBindings = {} or initialValue ≠ undefined) and regionalFrame □ PARAMETERFRAME and
         |regionalEnv| ≥ 2 then
      regionalFrame □ regionalEnv[|regionalEnv| – 2];
      existingBindings □ {b | □b □ regionalFrame.localBindings such that b.qname = qname}
   end if;
   if existingBindings = {} then
      v: DYNAMICVAR □ new DYNAMICVAR□□value: initialValue, sealed: true□□;
      regionalFrame.localBindings □ regionalFrame.localBindings □
            {LOCALBINDING□qname: qname, accesses: readWrite, content: v, explicit: false□};
      return v
   elsif |existingBindings| ≠ 1 then throw definitionError
   else
      b: LOCALBINDING □ the one element of existingBindings;
      m: LOCALMEMBER □ b.content;
      if b.accesses ≠ readWrite or m □ DYNAMICVAR or not m.sealed then
         throw definitionError
      end if;
      note At this point a hoisted binding of the same var already exists, so there is no need to create another one.
            Overwrite its initial value if the new definition is a function definition.
      if initialValue ≠ undefined then m.value □ initialValue end if;
      return m
   end if
end proc;
```

10.5.3 Adding Instance Definitions

```
proc searchForOverrides(c: CLASS, id: STRING, namespaces: NAMESPACE{}, access: ACCESS): VTABLEINDEX{}
   overriddenIndices: VTABLEINDEX{} □ {};
   s: CLASSOPT □ c.super;
   for each ns □ namespaces do
      qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: ns, id: id□;
      if s ≠ none then
         i: VTABLEINDEXOPT □ findVTableIndex(s, {qname}, access);
         if i ≠ none then overriddenIndices □ overriddenIndices □ {i} end if
      end if
   end for each;
   if |overriddenIndices| > 1 then throw definitionError end if;
   return overriddenIndices
end proc;

tuple OVERRIDESTATUS
   overriddenIndices: VTABLEINDEX{},
   definedMultiname: MULTINAME
end tuple;
```

```
proc checkOverrideConflicts(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE{}, access: ACCESS,
        m: INSTANCEMEMBER): OVERRIDESTATUS
    overriddenIndices: VTABLEINDEX{};
    definedMultiname: MULTINAME;
    if namespaces = {} then
        overriddenIndices ← searchForOverrides(c, id, cxt.openNamespaces, access);
        if overriddenIndices = {} then
            definedMultiname ← {QUALIFIEDNAME⟨namespace: publicNamespace, id: id⟩}
        else definedMultiname ← {}
        end if
    else
        overriddenIndices ← searchForOverrides(c, id, namespaces, access);
        definedMultiname ← {QUALIFIEDNAME⟨namespace: ns, id: id⟩ | ∀ns ∈ namespaces}
    end if;
    if some b ∈ c.instanceBindings satisfies b.qname ∈ definedMultiname and b.access = access then
        throw definitionError
    end if;
    for each i ∈ overriddenIndices do
        if some ve ∈ c.vTable satisfies ve.index = i then
            Throw an error because the same member cannot be overridden twice in the same class
            throw definitionError
        end if;
        mOverridden: INSTANCEMEMBER ← getInstanceMember(c, i);
        if mOverridden.final then throw definitionError end if;
        overridingAMethod: BOOLEAN ← mOverridden ∈ INSTANCEMETHOD;
        definingAMethod: BOOLEAN ← m ∈ INSTANCEMETHOD;
        if definingAMethod ≠ overridingAMethod then throw definitionError end if
    end for each;
    return OVERRIDESTATUS⟨overriddenIndices: overriddenIndices, definedMultiname: definedMultiname⟩
end proc;

proc defineInstanceMemberHalf(c: CLASS, os: OVERRIDESTATUS, access: ACCESS, m: INSTANCEMEMBER)
    index: VTABLEINDEX;
    if os.overriddenIndices = {} then index ← new VTABLEINDEX⟨⟨⟩⟩
    else index ← the one element of (os.overriddenIndices)
    end if;
    c.vTable ← c.vTable ∪ {VTABLEENTRY⟨index: index, content: m⟩};
    c.instanceBindings ← c.instanceBindings ∪ {INSTANCEBINDING⟨qname: qname, access: access, index: index⟩ |
            ∀qname ∈ os.definedMultiname}
end proc;
```

```
proc defineInstanceMember(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE{},
      overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, accesses: ACCESSSET, m: INSTANCEMEMBER):
      VTABLEINDEX{}
   if explicit then throw definitionError end if;
   readStatus: OVERRIDESTATUS ← OVERRIDESTATUS⟨⟨overriddenIndices: {}, definedMultiname: {}⟩⟩;
   writeStatus: OVERRIDESTATUS ← OVERRIDESTATUS⟨⟨overriddenIndices: {}, definedMultiname: {}⟩⟩;
   if accesses ∈ {read, readWrite} then
      readStatus ← checkOverrideConflicts(c, cxt, id, namespaces, read, m)
   end if;
   if accesses ∈ {write, readWrite} then
      writeStatus ← checkOverrideConflicts(c, cxt, id, namespaces, write, m)
   end if;
   overriddenIndices: VTABLEINDEX{} ← readStatus.overriddenIndices ∪ writeStatus.overriddenIndices;
   case overrideMod of
      {none} do
         if overriddenIndices ≠ {} then throw definitionError end if;
         warnedNamespaces: NAMESPACE{} ← {};
         if namespaces ≠ {} then warnedNamespaces ← cxt.openNamespaces – namespaces
         end if;
         if (accesses ∈ {read, readWrite} and searchForOverrides(c, id, warnedNamespaces, read) ≠ {}) or
               (accesses ∈ {write, readWrite} and searchForOverrides(c, id, warnedNamespaces, write) ≠ {}) then
            throw definitionError
         end if;
      {false} do if overriddenIndices ≠ {} then throw definitionError end if;
      {true} do if overriddenIndices = {} then throw definitionError end if;
      {undefined} do nothing
   end case;
   if accesses ∈ {read, readWrite} then
      defineInstanceMemberHalf(c, readStatus, read, m)
   end if;
   if accesses ∈ {write, readWrite} then
      defineInstanceMemberHalf(c, writeStatus, write, m)
   end if;
   return overriddenIndices
end proc;
```

10.5.4 Instantiation

```
proc instantiateFunction(uf: UNINSTANTIATEDFUNCTION, env: ENVIRONMENT): SIMPLEINSTANCE
    slots: SLOT{} □ {new SLOT□□id: s.id, value: s.value□□| □s □ uf.defaultSlots};
    localBindings: LOCALBINDING{} □ {};
    sealed: BOOLEAN;
    parent: OBJECTOPT □ none;
    if uf.buildPrototype then sealed □ false; ???? else sealed □ true end if;
    i: SIMPLEINSTANCE □ new SIMPLEINSTANCE□□localBindings: localBindings, parent: parent, sealed: sealed,
            type: uf.type, slots: slots, call: uf.call, construct: uf.construct, toClass: none, env: env□□;
    instantiations: SIMPLEINSTANCE{} □ uf.instantiations;
    if instantiations ≠ {} then
        Suppose that instantiateFunction were to choose at its discretion some element i2 of instantiations, assign
                i2.env □ env, and return i. If the behaviour of doing that assignment were observationally indistinguishable
                by the rest of the program from the behaviour of returning i without modifying i2.env, then the
                implementation may, but does not have to, return i2 now, discarding (or not even bothering to create) the
                value of i.
        note The above rule allows an implementation to avoid creating a fresh closure each time a local function is
                instantiated if it can show that the closures would behave identically. This optimisation is not transparent to
                the programmer because the instantiations will be === to each other and share one set of properties (including
                the prototype property, if applicable) rather than each having its own. ECMAScript programs should not
                rely on this distinction.
    end if;
    uf.instantiations □ instantiations □ {i};
    return i
end proc;

proc instantiateMember(m: LOCALMEMBER, env: ENVIRONMENT): LOCALMEMBER
    case m of
        {forbidden} □ CONSTRUCTORMETHOD do return m;
        VARIABLE do
            value: VARIABLEVALUE □ m.value;
            if value □ UNINSTANTIATEDFUNCTION then
                value □ instantiateFunction(value, env)
            end if;
            return new VARIABLE□□type: m.type, value: value, immutable: m.immutable□□
        DYNAMICVAR do
            value: OBJECT □ UNINSTANTIATEDFUNCTION □ m.value;
            if value □ UNINSTANTIATEDFUNCTION then
                value □ instantiateFunction(value, env)
            end if;
            return new DYNAMICVAR□□value: value, sealed: m.sealed□□
        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 MemberTranslation
   pluralMember: LocalMember,
   singularMember: LocalMember
end tuple;

proc instantiateBlockFrame(pluralFrame: BlockFrame, env: Environment): BlockFrame
   singularFrame: BlockFrame ← new BlockFrame⟨⟨localBindings: {}, plurality: singular⟩⟩;
   pluralMembers: LocalMember{} ← {b.content | ∀b ∈ pluralFrame.localBindings};
   memberTranslations: MemberTranslation{} ← {MemberTranslation⟨⟨pluralMember: m,
         singularMember: instantiateMember(m, [singularFrame] ⊕ env)⟩⟩ | ∀m ∈ pluralMembers};
   proc translateMember(m: LocalMember): LocalMember
      mi: MemberTranslation ← the one element mi ∈ memberTranslations that satisfies mi.pluralMember = m;
      return mi.singularMember
   end proc;
   singularFrame.localBindings ← {LocalBinding⟨⟨content: translateMember(b.content), other fields from b⟩⟩ |
         ∀b ∈ pluralFrame.localBindings};
   return singularFrame
end proc;

proc instantiateParameterFrame(pluralFrame: ParameterFrame, env: Environment, singularThis: ObjectOpt):
      ParameterFrame
   singularFrame: ParameterFrame ← new ParameterFrame⟨⟨localBindings: {}, plurality: singular,
         this: singularThis, unchecked: pluralFrame.unchecked, prototype: pluralFrame.prototype,
         restAllowsNames: pluralFrame.restAllowsNames, returnType: pluralFrame.returnType⟩⟩;
   pluralMembers: LocalMember{} ← {b.content | ∀b ∈ pluralFrame.localBindings};
   memberTranslations: MemberTranslation{} ← {MemberTranslation⟨⟨pluralMember: m,
         singularMember: instantiateMember(m, [singularFrame] ⊕ env)⟩⟩ | ∀m ∈ pluralMembers};
   proc translateMember(m: LocalMember): LocalMember
      mi: MemberTranslation ← the one element mi ∈ memberTranslations that satisfies mi.pluralMember = m;
      return mi.singularMember
   end proc;
   singularFrame.localBindings ← {LocalBinding⟨⟨content: translateMember(b.content), other fields from b⟩⟩ |
         ∀b ∈ pluralFrame.localBindings};
   singularFrame.positional ← [Parameter⟨⟨var: translateMember(op.var), default: op.default⟩⟩ |
         ∀op ∈ pluralFrame.positional];
   singularFrame.named ← {NamedParameter⟨⟨var: translateMember(np.var), other fields from np⟩⟩ |
         ∀np ∈ pluralFrame.named};
   if pluralFrame.rest = none then singularFrame.rest ← none
   else singularFrame.rest ← translateMember(pluralFrame.rest)
   end if;
   return singularFrame
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 ∪ GlobalObject ← getPackageOrGlobalFrame(env);
        if g ∈ GlobalObject then
            note 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 findLocalMember(o: FRAME ∪ SIMPLEINSTANCE ∪ REGEXP ∪ DATE, multiname: MULTINAME, access: ACCESS):
      LOCALMEMBEROPT
  matchingLocalBindings: LOCALBINDING{} ← {b | ∀b ∈ o.localBindings such that
        b.qname ∈ multiname and accessesOverlap(b.accesses, access)};
  note If the same member was found via several different bindings b, then it will appear only once in the set
        matchingLocalMembers.
  matchingLocalMembers: LOCALMEMBER{} ← {b.content | ∀b ∈ matchingLocalBindings};
  if matchingLocalMembers = {} then return none
  elsif |matchingLocalMembers| = 1 then return the one element of matchingLocalMembers
  else
    note This access is ambiguous because the bindings it found belong to several different local members.
    throw propertyAccessError
  end if
end proc;

proc findLocalVTableIndex(c: CLASS, multiname: MULTINAME, access: ACCESS): VTABLEINDEXOPT
  matchingIndices: VTABLEINDEX{} ← {b.index | ∀b ∈ c.instanceBindings such that
        b.qname ∈ multiname and b.access = access};
  note If the same INSTANCEMEMBER was found via several different bindings b, then it will appear only once in the set
        matchingIndices.
  if matchingIndices = {} then return none
  elsif |matchingIndices| = 1 then return the one element of matchingIndices
  else
    note This access is ambiguous because the bindings it found belong to several different members in the same class.
    throw propertyAccessError
  end if
end proc;

proc findCommonMember(o: OBJECT, multiname: MULTINAME, access: ACCESS, flat: BOOLEAN):
      {none} ∪ LOCALMEMBER ∪ VTABLEINDEX
  m: {none} ∪ LOCALMEMBER ∪ VTABLEINDEX;
  case o of
    UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHARACTER ∪ STRING ∪
          NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ METHODCLOSURE do
      return none;
    PACKAGE do return findLocalMember(o, multiname, access);
    SIMPLEINSTANCE ∪ REGEXP ∪ DATE ∪ GLOBALOBJECT do
      m ← findLocalMember(o, multiname, access);
    CLASS do
      m ← findLocalMember(o, multiname, access);
      if m = none then m ← findLocalVTableIndex(o, multiname, access) end if
  end case;
  if m ≠ none then return m end if;
  parent: OBJECTOPT ← o.parent;
  if parent ≠ none then
    m ← findCommonMember(parent, multiname, access, flat);
    if flat and m ∈ DYNAMICVAR then m ← none end if
  end if;
  return m
end proc;
```

```
proc findVTableIndex(c: CLASS, multiname: MULTINAME, access: ACCESS): VTABLEINDEXOPT
    note Start from the root class (Object) and proceed through more specific classes that are ancestors of c.
    for each s □ ancestors(c) do
        i: VTABLEINDEXOPT □ findLocalVTableIndex(s, multiname, access);
        if i ≠ none then return i end if
    end for each;
    return none
end proc;
```

getInstanceMember(*c*, *index*) returns the instance member at location *index* in class *c*'s vtable. The caller of *getInstanceMember* ensures that such a member always exists.

```
proc getInstanceMember(c: CLASS, index: VTABLEINDEX): INSTANCEMEMBER
    if some ve □ c.vTable satisfies ve.index = index then return ve.content
    else return getInstanceMember(c.super, index)
    end if
end proc;

proc lookupInstanceMember(c: CLASS, qname: QUALIFIEDNAME, access: ACCESS): INSTANCEMEMBEROPT
    index: VTABLEINDEXOPT □ findVTableIndex(c, {qname}, access);
    if index = none then return none end if;
    return getInstanceMember(c, index)
end proc;
```

10.5.7 Reading a Property

```
proc readProperty(container: OBJOPTIONALLIMIT ∪ FRAME, multiname: MULTINAME, kind: LOOKUPKIND,
      phase: PHASE): OBJECTOPT
   case container of
      OBJECT do
         c: CLASS ← objectType(container);
         index: VTABLEINDEXOPT ← findVTableIndex(c, multiname, read);
         if index ≠ none then return readInstanceMember(container, c, index, phase)
         end if;
         m: {none} ∪ LOCALMEMBER ∪ VTABLEINDEX ← findCommonMember(container, multiname, read, false);
         case m of
            {none} do
               if kind = propertyLookup and
                     container ∈ SIMPLEINSTANCE ∪ DATE ∪ REGEXP ∪ GLOBALOBJECT and not container.sealed and
                     (some qname ∈ multiname satisfies qname.namespace = publicNamespace) then
                  case phase of
                     {compile} do throw compileExpressionError;
                     {run} do return undefined
                  end case
               else return none
               end if;
            LOCALMEMBER do return readLocalMember(m, phase);
            VTABLEINDEX do
               if container ∉ CLASS or kind = propertyLookup then
                  throw propertyAccessError
               end if;
               this: OBJECTIOPT ← kind.this;
               case this of
                  {none} do throw propertyAccessError;
                  {inaccessible} do throw compileExpressionError;
                  OBJECT do
                     return readInstanceMember(this, objectType(this), m, phase)
               end case
         end case;
      SYSTEMFRAME ∪ PARAMETERFRAME ∪ BLOCKFRAME do
         m: LOCALMEMBEROPT ← findLocalMember(container, multiname, read);
         if m = none then return none else return readLocalMember(m, phase) end if;
      LIMITEDINSTANCE do
         superclass: CLASSOPT ← container.limit.super;
         if superclass = none then return none end if;
         index: VTABLEINDEXOPT ← findVTableIndex(superclass, multiname, read);
         if index ≠ none then
            return readInstanceMember(container.instance, superclass, index, phase)
         else return none
         end if
   end case
end proc;
```

```
proc readInstanceMember(this: OBJECT, c: CLASS, index: VTABLEINDEX, phase: PHASE): OBJECT
    m: INSTANCEMEMBER ← getInstanceMember(c, index);
    case m of
        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 readLocalMember(m: LOCALMEMBER, phase: PHASE): OBJECT
    case m of
        {forbidden} do throw propertyAccessError;
        VARIABLE do
            if phase = compile and not m.immutable then throw compileExpressionError
            end if;
            value: VARIABLEVALUE ← m.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;
                UNINSTANTIATEDFUNCTION do
                    note An uninstantiated function can only be found when phase = compile.
                    throw compileExpressionError;
                () → OBJECT do
                    note phase = compile because all futures are resolved by the end of the compilation phase.
                    m.value ← inaccessible;
                    type: CLASS ← getVariableType(m, phase);
                    newValue: OBJECT ← value();
                    coercedValue: OBJECT ← type.implicitCoerce(newValue, false);
                    m.value ← coercedValue;
                    return newValue
            end case;
        DYNAMICVAR do
            if phase = compile then throw compileExpressionError end if;
            value: OBJECT ∪ UNINSTANTIATEDFUNCTION ← m.value;
            note value can be an UNINSTANTIATEDFUNCTION 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;
```

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
    OBJECT do
      c: CLASS ← objectType(container);
      index: VTABLEINDEXOPT ← findVTableIndex(c, multiname, write);
      if index ≠ none then
        writeInstanceMember(container, c, index, newValue, phase);
        return ok
      end if;
      m: {none} ∪ LOCALMEMBER ∪ VTABLEINDEX ← findCommonMember(container, multiname, write, true);
      case m of
        {none} do
          if createIfMissing and container ∈ SIMPLEINSTANCE ∪ DATE ∪ REGEXP ∪ GLOBALOBJECT and
                not container.sealed and (some qname ∈ multiname satisfies
                qname.namespace = publicNamespace) then
            note Before trying to create a new dynamic property named qname, check that there is no read-only
                fixed property with the same name.
            if findVTableIndex(c, {qname}, read) = none and
                  findCommonMember(container, {qname}, read, true) = none then
              dv: DYNAMICVAR ← new DYNAMICVAR⟨⟨value: newValue, sealed: false⟩⟩;
              container.localBindings ← container.localBindings ∪ {LOCALBINDING⟨⟨qname: qname,
                  accesses: readWrite, content: dv, explicit: false⟩⟩};
              return ok
            end if
          end if;
          return none;
        LOCALMEMBER do writeLocalMember(m, newValue, phase); return ok;
        VTABLEINDEX do
          if container ∉ CLASS or kind = propertyLookup then
            throw propertyAccessError
          end if;
          note this cannot be inaccessible during the run phase.
          this: OBJECTOPT ← kind.this;
          case this of
            {none} do throw propertyAccessError;
            OBJECT do
              writeInstanceMember(this, objectType(this), m, newValue, phase);
              return ok
          end case
      end case;
    SYSTEMFRAME ∪ PARAMETERFRAME ∪ BLOCKFRAME do
      m: LOCALMEMBEROPT ← findLocalMember(container, multiname, write);
      if m = none then return none
      else writeLocalMember(m, newValue, phase); return ok
      end if;
    LIMITEDINSTANCE do
      superclass: CLASSOPT ← container.limit.super;
      if superclass = none then return none end if;
      index: VTABLEINDEXOPT ← findVTableIndex(superclass, multiname, write);
      if index ≠ none then
        writeInstanceMember(container.instance, superclass, index, newValue, phase);
        return ok
      else return none
      end if
    end case
end proc;

proc writeInstanceMember(this: OBJECT, c: CLASS, index: VTABLEINDEX, newValue: OBJECT, phase: {run})
    m: INSTANCEMEMBER  getInstanceMember(c, index);
    case m of
        INSTANCEVARIABLE do
            s: SLOT  findSlot(this, m);
            if m.immutable and s.value ≠ uninitialised then throw propertyAccessError
            end if;
            coercedValue: OBJECT  m.type.implicitCoerce(newValue, false);
            s.value  coercedValue;
        INSTANCEMETHOD do throw propertyAccessError;
        INSTANCEGETTER do
            m cannot be an INSTANCEGETTER because these are only represented as read-only members.
        INSTANCESETTER do
            coercedValue: OBJECT  m.type.implicitCoerce(newValue, false);
            m.call(this, coercedValue, m.env, phase)
    end case
end proc;

proc writeLocalMember(m: LOCALMEMBER, newValue: OBJECT, phase: {run})
    case m of
        {forbidden}  CONSTRUCTORMETHOD do throw propertyAccessError;
        VARIABLE do
            if m.value = inaccessible or (m.immutable and m.value ≠ uninitialised) then
                throw propertyAccessError
            end if;
            type: CLASS  getVariableType(m, phase);
            coercedValue: OBJECT  type.implicitCoerce(newValue, false);
            m.value  coercedValue;
        DYNAMICVAR do m.value  newValue;
        GETTER do
            m cannot be a GETTER because these are only represented as read-only members.
        SETTER do
            coercedValue: OBJECT  m.type.implicitCoerce(newValue, false);
            env: ENVIRONMENTI  m.env;
            note All instances are resolved for the run phase, so env ≠ inaccessible.
            m.call(coercedValue, env, phase)
    end case
end proc;
```

```
proc getVariableType(v: VARIABLE, phase: PHASE): CLASS
    type: VARIABLETYPE □ v.type;
    case type of
        CLASS do return type;
        {inaccessible} do
            note 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 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;
```

10.5.9 Deleting a Property

```
proc deleteProperty(container: ObjOptionalLimit □ Frame, multiname: Multiname, kind: LookupKind,
        phase: {run}): BooleanOpt
    case container of
        Object do
            c: Class □ objectType(container);
            if findVTableIndex(c, multiname, write) ≠ none then return false end if;
            m: {none} □ LocalMember □ VTableIndex □ findCommonMember(container, multiname, write, true);
            case m of
                {none} do return none;
                {forbidden} do throw propertyAccessError;
                Variable □ ConstructorMethod □ Getter □ Setter do return false;
                DynamicVar do
                    if m.sealed then return false
                    else
                        container.localBindings □ {b | □b □ container.localBindings such that
                                b.qname □ multiname or b.content ≠ m};
                        return true
                    end if;
                VTableIndex do
                    if container □ Class or kind = propertyLookup then return false
                    end if;
                    note this cannot be inaccessible during the run phase.
                    this: ObjectOpt □ kind.this;
                    case this of
                        {none} do throw propertyAccessError;
                        Object do return false
                    end case
            end case;
        SystemFrame □ ParameterFrame □ BlockFrame do
            if findLocalMember(container, multiname, write) ≠ none then return false
            else return none
            end if;
        LimitedInstance do
            superclass: ClassOpt □ container.limit.super;
            if superclass = none then return none end if;
            if findVTableIndex(superclass, multiname, write) ≠ none then return false
            else return none
            end if
    end case
end proc;
```

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 Setup on the goal nonterminal, which will recursively call Setup on some intermediate nonterminals.
- 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 □:

□□ {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

```
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);
    Setup[ParenExpression]();
    q: OBJECT □ readReference(Eval[ParenExpression](env, compile), 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;
```

Setup

```
proc Setup[SimpleQualifiedIdentifier] ()
    [SimpleQualifiedIdentifier □ Identifier] do nothing;
    [SimpleQualifiedIdentifier □ Qualifier :: Identifier] do nothing
end proc;

proc Setup[ExpressionQualifiedIdentifier □ ParenExpression :: Identifier] ()
end proc;
```

Setup[*QualifiedIdentifier*] () propagates the call to Setup to every nonterminal in the expansion of *QualifiedIdentifier*.

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 Validate[ArrayLiteral](cxt, env);
    [PrimaryExpression □ ObjectLiteral] do Validate[ObjectLiteral](cxt, env);
    [PrimaryExpression □ FunctionExpression] do Validate[FunctionExpression](cxt, env)
end proc;
```

Validate[*ParenExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ParenExpression*.

Validate[*ParenListExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ParenListExpression*.

Setup

Setup[*PrimaryExpression*] () propagates the call to Setup to every nonterminal in the expansion of *PrimaryExpression*.

Setup[*ParenExpression*] () propagates the call to Setup to every nonterminal in the expansion of *ParenExpression*.

Setup[*ParenListExpression*] () propagates the call to Setup to every nonterminal in the expansion of *ParenListExpression*.

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 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 return Eval[ArrayLiteral](env, phase);
    [PrimaryExpression □ ObjectLiteral] do return Eval[ObjectLiteral](env, phase);
    [PrimaryExpression □ FunctionExpression] do
        return Eval[FunctionExpression](env, phase)
end proc;

proc Eval[ParenExpression □ ( AssignmentExpression^allowIn )] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    return Eval[AssignmentExpression^allowIn](env, phase)
end proc;

proc Eval[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ParenListExpression □ ParenExpression] do return Eval[ParenExpression](env, phase);
    [ParenListExpression □ ( ListExpression^allowIn , AssignmentExpression^allowIn )] do
        readReference(Eval[ListExpression^allowIn](env, phase), phase);
        return readReference(Eval[AssignmentExpression^allowIn](env, phase), phase)
end proc;

proc EvalAsList[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
    [ParenListExpression □ ParenExpression] do
        elt: OBJECT □ readReference(Eval[ParenExpression](env, phase), phase);
        return [elt];
    [ParenListExpression ⇒ ( ListExpression^allowIn , AssignmentExpression^allowIn )] do
        elts: OBJECT[] ← EvalAsList[ListExpression^allowIn](env, phase);
        elt: OBJECT ← readReference(Eval[AssignmentExpression^allowIn](env, phase), phase);
        return elts ⊕ [elt]
end proc;
```

12.4 Function Expressions

Syntax

```
FunctionExpression ⇒
      function FunctionCommon
   |  function Identifier FunctionCommon
```

Validation

```
F[FunctionExpression]: UNINSTANTIATEDFUNCTION;

proc Validate[FunctionExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [FunctionExpression ⇒ function FunctionCommon] do
        unchecked: BOOLEAN ← not cxt.strict and Plain[FunctionCommon];
        this: {none, inaccessible} ← unchecked ? inaccessible : none;
        F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](cxt, env, this, unchecked, unchecked);
    [FunctionExpression ⇒ function Identifier FunctionCommon] do
        v: VARIABLE ← new VARIABLE⟨⟨type: classClass, value: inaccessible, immutable: true⟩⟩;
        b: LOCALBINDING ← LOCALBINDING⟨⟨qname:
                QUALIFIEDNAME⟨⟨namespace: publicNamespace, id: Name[Identifier]⟩⟩, accesses: readWrite, content: v,
                explicit: false⟩⟩;
        compileFrame: BLOCKFRAME ← new BLOCKFRAME⟨⟨localBindings: {b}, plurality: plural⟩⟩;
        unchecked: BOOLEAN ← not cxt.strict and Plain[FunctionCommon];
        this: {none, inaccessible} ← unchecked ? inaccessible : none;
        F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](cxt, [compileFrame] ⊕ env, this, unchecked,
                unchecked)
end proc;
```

Setup

```
proc Setup[FunctionExpression] ()
    [FunctionExpression ⇒ function FunctionCommon] do Setup[FunctionCommon]();
    [FunctionExpression ⇒ function Identifier FunctionCommon] do Setup[FunctionCommon]()
end proc;
```

Evaluation

```
proc Eval[FunctionExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [FunctionExpression ⇒ function FunctionCommon] do
        if phase = compile then throw compileExpressionError end if;
        return instantiateFunction(F[FunctionExpression], env);
    [FunctionExpression → function Identifier FunctionCommon] do
       if phase = compile then throw compileExpressionError end if;
       v: VARIABLE ← new VARIABLE⟨⟨type: classClass, value: inaccessible, immutable: true⟩⟩;
       b: LOCALBINDING ← LOCALBINDING⟨⟨qname:
             QUALIFIEDNAME⟨⟨namespace: publicNamespace, id: Name[Identifier]⟩⟩, accesses: readWrite, content: v,
             explicit: false⟩⟩;
       runtimeFrame: BLOCKFRAME ← new BLOCKFRAME⟨⟨localBindings: {b}, plurality: plural⟩⟩;
       f2: SIMPLEINSTANCE ← instantiateFunction(F[FunctionExpression], [runtimeFrame] ⊕ env);
       v.value ← f2;
       return f2
 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 {toString(Value[Number], compile)}
end proc;
```

Setup

Setup[*ObjectLiteral*] () propagates the call to Setup to every nonterminal in the expansion of *ObjectLiteral*.

Setup[*FieldList*] () propagates the call to Setup to every nonterminal in the expansion of *FieldList*.

Setup[*LiteralField*] () propagates the call to Setup to every nonterminal in the expansion of *LiteralField*.

```
proc Setup[FieldName] ()
    [FieldName □ Identifier] do nothing;
    [FieldName □ String] do nothing;
    [FieldName □ Number] do nothing
end proc;
```

Evaluation

```
proc Eval[ObjectLiteral] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ObjectLiteral □ { }] do
        if phase = compile then throw compileExpressionError end if;
        return new SIMPLEINSTANCE□localBindings: {}, parent: objectPrototype, sealed: false, type: prototypeClass,
                slots: {}, call: none, construct: none, toClass: none, env: none□
    [ObjectLiteral □ { FieldList }] do
        if phase = compile then throw compileExpressionError end if;
        localBindings: LOCALBINDING{} □ Eval[FieldList](env, phase);
        return new SIMPLEINSTANCE□localBindings: localBindings, parent: objectPrototype, sealed: false,
                type: prototypeClass, slots: {}, call: none, construct: none, toClass: none, env: none□
end proc;

proc Eval[FieldList] (env: ENVIRONMENT, phase: PHASE): LOCALBINDING{}
    [FieldList □ LiteralField] do
        na: NAMEDARGUMENT □ Eval[LiteralField](env, phase);
        qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: publicNamespace, id: na.name□
        p: DYNAMICVAR □ new DYNAMICVAR□value: na.value, sealed: false□
        return {LOCALBINDING□qname: qname, accesses: readWrite, content: p, explicit: false□};
    [FieldList_0 □ FieldList_1 , LiteralField] do
        localBindings: LOCALBINDING{} □ Eval[FieldList_1](env, phase);
        na: NAMEDARGUMENT □ Eval[LiteralField](env, phase);
        qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: publicNamespace, id: na.name□
        if some b □ localBindings satisfies b.qname = qname then
            throw argumentMismatchError
        end if;
        p: DYNAMICVAR □ new DYNAMICVAR□value: na.value, sealed: false□
        return localBindings □ {LOCALBINDING□qname: qname, accesses: readWrite, content: p, explicit: false□}
end proc;
```

```
proc Eval[LiteralField  FieldName : AssignmentExpression^allowIn]
      (env: ENVIRONMENT, phase: PHASE): NAMEDARGUMENT
   name: STRING  Eval[FieldName](env, phase);
   value: OBJECT  readReference(Eval[AssignmentExpression^allowIn](env, phase), phase);
   return NAMEDARGUMENTname: 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 toString(Value[Number], compile)
end proc;
```

12.6 Array Literals

Syntax

```
ArrayLiteral  [ ElementList ]

ElementList 
      LiteralElement
   |  ElementList , LiteralElement

LiteralElement 
      «empty»
   |  AssignmentExpression^allowIn
```

Validation

```
proc Validate[ArrayLiteral  [ ElementList ]] (cxt: CONTEXT, env: ENVIRONMENT)
   ????
end proc;
```

Setup

```
proc Setup[ArrayLiteral  [ ElementList ]] ()
   ????
end proc;
```

Evaluation

```
proc Eval[ArrayLiteral  [ ElementList ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   ????
end proc;
```

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;
```

Setup

Setup[*SuperExpression*] () propagates the call to Setup to every nonterminal in the expansion of *SuperExpression*.

Evaluation

```
proc Eval[SuperExpression] (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT
   [SuperExpression → super] do
      this: OBJECTIOPT ← findThis(env, false);
      note Validate ensured that this cannot be none at this point.
      if this = inaccessible then throw compileExpressionError end if;
      limit: CLASSOPT ← getEnclosingClass(env);
      note 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 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 not limit.isInstanceOf(o) 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 MemberOperator
   |  FullPostfixExpression Arguments
   |  PostfixExpression [no line break] ++
   |  PostfixExpression [no line break] --

FullNewExpression □  new FullNewSubexpression Arguments

FullNewSubexpression □
      PrimaryExpression
   |  QualifiedIdentifier
   |  FullNewExpression
   |  FullNewSubexpression MemberOperator
   |  SuperExpression MemberOperator

ShortNewExpression □  new ShortNewSubexpression

ShortNewSubexpression □
      FullNewSubexpression
   |  ShortNewExpression
```

Validation

Validate[*PostfixExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *PostfixExpression*.

```
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 MemberOperator] do
        Validate[SuperExpression](cxt, env);
        Validate[MemberOperator](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;
```

Validate[*FullNewExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *FullNewExpression*.

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);
    [FullNewSubexpression$_0$ → FullNewSubexpression$_1$ MemberOperator] do
        Validate[FullNewSubexpression$_1$](cxt, env);
        Validate[MemberOperator](cxt, env);
    [FullNewSubexpression → SuperExpression MemberOperator] do
        Validate[SuperExpression](cxt, env);
        Validate[MemberOperator](cxt, env)
end proc;
```

Validate[*ShortNewExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ShortNewExpression*.

Validate[*ShortNewSubexpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ShortNewSubexpression*.

Setup

Setup[*PostfixExpression*] () propagates the call to Setup to every nonterminal in the expansion of *PostfixExpression*.

Setup[*AttributeExpression*] () propagates the call to Setup to every nonterminal in the expansion of *AttributeExpression*.

Setup[*FullPostfixExpression*] () propagates the call to Setup to every nonterminal in the expansion of *FullPostfixExpression*.

Setup[*FullNewExpression*] () propagates the call to Setup to every nonterminal in the expansion of *FullNewExpression*.

Setup[*FullNewSubexpression*] () propagates the call to Setup to every nonterminal in the expansion of *FullNewSubexpression*.

Setup[*ShortNewExpression*] () propagates the call to Setup to every nonterminal in the expansion of *ShortNewExpression*.

Setup[*ShortNewSubexpression*] () propagates the call to Setup to every nonterminal in the expansion of *ShortNewSubexpression*.

Evaluation

```
proc Eval[PostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [PostfixExpression □ AttributeExpression] do
      return Eval[AttributeExpression](env, phase);
   [PostfixExpression □ FullPostfixExpression] do
      return Eval[FullPostfixExpression](env, phase);
   [PostfixExpression □ ShortNewExpression] do
      return Eval[ShortNewExpression](env, phase)
end proc;

proc Eval[AttributeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [AttributeExpression □ SimpleQualifiedIdentifier] do
      return LEXICALREFERENCE□env: env, variableMultiname: Multiname[SimpleQualifiedIdentifier],
            strict: Strict[AttributeExpression]□
   [AttributeExpression$_0$ □ AttributeExpression$_1$ MemberOperator] do
      a: OBJECT □ readReference(Eval[AttributeExpression$_1$](env, phase), phase);
      return Eval[MemberOperator](env, a, phase);
   [AttributeExpression$_0$ □ AttributeExpression$_1$ Arguments] do
      r: OBJORREF □ Eval[AttributeExpression$_1$](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);
   [FullPostfixExpression$_0$ □ FullPostfixExpression$_1$ MemberOperator] do
      a: OBJECT □ readReference(Eval[FullPostfixExpression$_1$](env, phase), phase);
      return Eval[MemberOperator](env, a, phase);
   [FullPostfixExpression □ SuperExpression MemberOperator] do
      a: OBJOPTIONALLIMIT □ Eval[SuperExpression](env, phase);
      return Eval[MemberOperator](env, a, phase);
   [FullPostfixExpression$_0$ □ FullPostfixExpression$_1$ Arguments] do
      r: OBJORREF □ Eval[FullPostfixExpression$_1$](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, 1.0f64, 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, 1.0f64, phase);
        writeReference(r, c, phase);
        return b
end proc;

proc Eval[FullNewExpression □ new FullNewSubexpression Arguments]
        (env: ENVIRONMENT, phase: PHASE): OBJORREF
    f: OBJECT □ readReference(Eval[FullNewSubexpression](env, phase), 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
        a: OBJECT □ readReference(Eval[FullNewSubexpression1](env, phase), phase);
        return Eval[MemberOperator](env, a, phase);
    [FullNewSubexpression □ SuperExpression MemberOperator] do
        a: OBJOPTIONALLIMIT □ Eval[SuperExpression](env, phase);
        return Eval[MemberOperator](env, a, phase)
end proc;

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

proc Eval[ShortNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ShortNewSubexpression □ FullNewSubexpression] do
        return Eval[FullNewSubexpression](env, phase);
    [ShortNewSubexpression □ ShortNewExpression] do
        return Eval[ShortNewExpression](env, phase)
end proc;
```

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 □ GENERALNUMBER □ CHARACTER □ STRING □ NAMESPACE □
                COMPOUNDATTRIBUTE □ DATE □ REGEXP □ PACKAGE □ GLOBALOBJECT do
            throw badValueError;
        CLASS do return a.call(this, args, phase);
        SIMPLEINSTANCE do
            f: OBJECT □ ARGUMENTLIST □ ENVIRONMENT □ PHASE □  OBJECT □ {none} □  a.call;
            if f = none then throw badValueError end if;
            return f(this, args, a.env, phase);
        METHODCLOSURE do return a.method.call(a.this, args, phase)
    end case
end proc;

proc construct(a: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
    case a of
        UNDEFINED □ NULL □ BOOLEAN □ GENERALNUMBER □ CHARACTER □ STRING □ NAMESPACE □
                COMPOUNDATTRIBUTE □ METHODCLOSURE □ DATE □ REGEXP □ PACKAGE □ GLOBALOBJECT do
            throw badValueError;
        CLASS do return a.construct(args, phase);
        SIMPLEINSTANCE do
            f: ARGUMENTLIST □ ENVIRONMENT □ PHASE □  OBJECT □ {none} □  a.construct;
            if f = none then throw badValueError end if;
            return f(args, a.env, phase)
    end case
end proc;
```

12.9 Member Operators

Syntax

```
MemberOperator □
      . QualifiedIdentifier
   |  Brackets

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

Arguments □
      ParenExpressions
   |  ( NamedArgumentList )
```

```
ParenExpressions □
      ( )
   |  ParenListExpression

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

Validation

Validate[*MemberOperator*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *MemberOperator*.

```
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;
```

Validate[*ParenExpressions*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ParenExpressions*.

```
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;
```

Setup

Setup[*MemberOperator*] () propagates the call to Setup to every nonterminal in the expansion of *MemberOperator*.

Setup[*Brackets*] () propagates the call to Setup to every nonterminal in the expansion of *Brackets*.

Setup[*Arguments*] () propagates the call to Setup to every nonterminal in the expansion of *Arguments*.

Setup[*ParenExpressions*] () propagates the call to Setup to every nonterminal in the expansion of *ParenExpressions*.

Setup[*NamedArgumentList*] () propagates the call to Setup to every nonterminal in the expansion of *NamedArgumentList*.

Evaluation

```
proc Eval[MemberOperator] (env: ENVIRONMENT, base: OBJOPTIONALLIMIT, phase: PHASE): OBJORREF
  [MemberOperator → . QualifiedIdentifier] do
    return DOTREFERENCE⟨base: base, propertyMultiname: Multiname[QualifiedIdentifier]⟩
  [MemberOperator → 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
   |  - NegatedMinLong
   |  ~ 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;
   [UnaryExpression$_0$ □ void UnaryExpression$_1$] do Validate[UnaryExpression$_1$](cxt, env);
   [UnaryExpression$_0$ □ typeof UnaryExpression$_1$] do
      Validate[UnaryExpression$_1$](cxt, env);
   [UnaryExpression □ ++ PostfixExpression] do Validate[PostfixExpression](cxt, env);
   [UnaryExpression □ -- PostfixExpression] do Validate[PostfixExpression](cxt, env);
   [UnaryExpression$_0$ □ + UnaryExpression$_1$] do Validate[UnaryExpression$_1$](cxt, env);
   [UnaryExpression$_0$ □ - UnaryExpression$_1$] do Validate[UnaryExpression$_1$](cxt, env);
   [UnaryExpression □ - NegatedMinLong] do nothing;
   [UnaryExpression$_0$ □ ~ UnaryExpression$_1$] do Validate[UnaryExpression$_1$](cxt, env);
   [UnaryExpression$_0$ □ ! UnaryExpression$_1$] do Validate[UnaryExpression$_1$](cxt, env)
end proc;
```

Setup

Setup[*UnaryExpression*] () propagates the call to Setup to every nonterminal in the expansion of *UnaryExpression*.

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);
   [UnaryExpression$_0$ □ void UnaryExpression$_1$] do
      readReference(Eval[UnaryExpression$_1$](env, phase), phase);
      return undefined;
```

```
  [UnaryExpression$_0$ → typeof UnaryExpression$_1$] do
     a: OBJECT ← readReference(Eval[UnaryExpression$_1$](env, phase), phase);
     c: CLASS ← objectType(a);
     return c.typeofString;
  [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$_{f64}$, 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$_{f64}$, phase);
     writeReference(r, c, phase);
     return c;
  [UnaryExpression$_0$ → + UnaryExpression$_1$] do
     a: OBJECT ← readReference(Eval[UnaryExpression$_1$](env, phase), phase);
     return plus(a, phase);
  [UnaryExpression$_0$ → - UnaryExpression$_1$] do
     a: OBJECT ← readReference(Eval[UnaryExpression$_1$](env, phase), phase);
     return minus(a, phase);
  [UnaryExpression → - NegatedMinLong] do return LONG⟨value: $-2^{63}$⟩;
  [UnaryExpression$_0$ → ~ UnaryExpression$_1$] do
     a: OBJECT ← readReference(Eval[UnaryExpression$_1$](env, phase), phase);
     return bitNot(a, phase);
  [UnaryExpression$_0$ → ! UnaryExpression$_1$] do
     a: OBJECT ← readReference(Eval[UnaryExpression$_1$](env, phase), 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);
  return generalNumberNegate(x)
end proc;

proc generalNumberNegate(x: GENERALNUMBER): GENERALNUMBER
  case x of
     LONG do return integerToLong(–(x.value));
     ULONG do return integerToULong(–(x.value));
     FLOAT32 do return float32Negate(x);
     FLOAT64 do return float64Negate(x)
  end case
end proc;
```

```
proc bitNot(a: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER □ toGeneralNumber(a, phase);
    case x of
        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)□;
        FLOAT32 □ FLOAT64 do
            i: {$-2^{31}$ ... $2^{31} - 1$} □ signedWrap32(truncateToInteger(x));
            return realToFloat64(bitwiseXor(i, –1))
    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

Validate[*MultiplicativeExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *MultiplicativeExpression*.

Setup

Setup[*MultiplicativeExpression*] () propagates the call to Setup to every nonterminal in the expansion of *MultiplicativeExpression*.

Evaluation

```
proc Eval[MultiplicativeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [MultiplicativeExpression □ UnaryExpression] do
        return Eval[UnaryExpression](env, phase);
    [$MultiplicativeExpression_0$ □ $MultiplicativeExpression_1$ * UnaryExpression] do
        a: OBJECT □ readReference(Eval[$MultiplicativeExpression_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[UnaryExpression](env, phase), phase);
        return multiply(a, b, phase);
    [$MultiplicativeExpression_0$ □ $MultiplicativeExpression_1$ / UnaryExpression] do
        a: OBJECT □ readReference(Eval[$MultiplicativeExpression_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[UnaryExpression](env, phase), phase);
        return divide(a, b, phase);
    [MultiplicativeExpression$_0$ $\Rightarrow$ MultiplicativeExpression$_1$ % UnaryExpression] do
        a: OBJECT $\leftarrow$ readReference(Eval[MultiplicativeExpression$_1$](env, phase), phase);
        b: OBJECT $\leftarrow$ readReference(Eval[UnaryExpression](env, phase), phase);
        return remainder(a, b, phase)
end proc;

proc multiply(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER $\leftarrow$ toGeneralNumber(a, phase);
    y: GENERALNUMBER $\leftarrow$ toGeneralNumber(b, phase);
    if x $\in$ LONG $\cup$ ULONG or y $\in$ LONG $\cup$ ULONG then
        i: INTEGEROPT $\leftarrow$ checkInteger(x);
        j: INTEGEROPT $\leftarrow$ checkInteger(y);
        if i $\neq$ none and j $\neq$ none then
            k: INTEGER $\leftarrow$ $i \cdot j$;
            if x $\in$ ULONG or y $\in$ ULONG then return integerToULong(k)
            else return integerToLong(k)
            end if
        end if
    end if;
    return float64Multiply(toFloat64(x), toFloat64(y))
end proc;

proc divide(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER $\leftarrow$ toGeneralNumber(a, phase);
    y: GENERALNUMBER $\leftarrow$ toGeneralNumber(b, phase);
    if x $\in$ LONG $\cup$ ULONG or y $\in$ LONG $\cup$ ULONG then
        i: INTEGEROPT $\leftarrow$ checkInteger(x);
        j: INTEGEROPT $\leftarrow$ checkInteger(y);
        if i $\neq$ none and j $\neq$ none and $j \neq 0$ then
            q: RATIONAL $\leftarrow$ $i/j$;
            if x $\in$ ULONG or y $\in$ ULONG then return rationalToULong(q)
            else return rationalToLong(q)
            end if
        end if
    end if;
    return float64Divide(toFloat64(x), toFloat64(y))
end proc;

proc remainder(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER $\leftarrow$ toGeneralNumber(a, phase);
    y: GENERALNUMBER $\leftarrow$ toGeneralNumber(b, phase);
    if x $\in$ LONG $\cup$ ULONG or y $\in$ LONG $\cup$ ULONG then
        i: INTEGEROPT $\leftarrow$ checkInteger(x);
        j: INTEGEROPT $\leftarrow$ checkInteger(y);
        if i $\neq$ none and j $\neq$ none and $j \neq 0$ then
            q: RATIONAL $\leftarrow$ $i/j$;
            k: INTEGER $\leftarrow$ $q \geq 0$ ? $\lfloor q \rfloor$ : $\lceil q \rceil$;
            r: INTEGER $\leftarrow$ $i - j \cdot k$;
            if x $\in$ ULONG or y $\in$ ULONG then return integerToULong(r)
            else return integerToLong(r)
            end if
        end if
    end if;
    return float64Remainder(toFloat64(x), toFloat64(y))
end proc;
```

12.12 Additive Operators

Syntax

```
AdditiveExpression □
     MultiplicativeExpression
  |  AdditiveExpression + MultiplicativeExpression
  |  AdditiveExpression - MultiplicativeExpression
```

Validation

Validate[*AdditiveExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *AdditiveExpression*.

Setup

Setup[*AdditiveExpression*] () propagates the call to Setup to every nonterminal in the expansion of *AdditiveExpression*.

Evaluation

```
proc Eval[AdditiveExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [AdditiveExpression □ MultiplicativeExpression] do
      return Eval[MultiplicativeExpression](env, phase);
   [AdditiveExpression$_0$ □ AdditiveExpression$_1$ + MultiplicativeExpression] do
      a: OBJECT □ readReference(Eval[AdditiveExpression$_1$](env, phase), phase);
      b: OBJECT □ readReference(Eval[MultiplicativeExpression](env, phase), phase);
      return add(a, b, phase);
   [AdditiveExpression$_0$ □ AdditiveExpression$_1$ - MultiplicativeExpression] do
      a: OBJECT □ readReference(Eval[AdditiveExpression$_1$](env, phase), phase);
      b: OBJECT □ readReference(Eval[MultiplicativeExpression](env, phase), 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) $\oplus$ 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: INTEGEROPT □ checkInteger(x);
      j: INTEGEROPT □ checkInteger(y);
      if $i \neq$ none and $j \neq$ none then
         k: INTEGER □ $i + j$;
         if x □ ULONG or y □ ULONG then return integerToULong(k)
         else return integerToLong(k)
         end if
      end if
   end if;
   return float64Add(toFloat64(x), toFloat64(y))
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: INTEGEROPT ← checkInteger(x);
      j: INTEGEROPT ← checkInteger(y);
      if i ≠ none and j ≠ none then
         k: INTEGER ← i – j;
         if x ∈ ULONG or y ∈ ULONG then return integerToULong(k)
         else return integerToLong(k)
         end if
      end if
   end if;
   return float64Subtract(toFloat64(x), toFloat64(y))
end proc;
```

12.13 Bitwise Shift Operators

Syntax

```
ShiftExpression ⇒
      AdditiveExpression
   |  ShiftExpression << AdditiveExpression
   |  ShiftExpression >> AdditiveExpression
   |  ShiftExpression >>> AdditiveExpression
```

Validation

Validate[*ShiftExpression*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ShiftExpression*.

Setup

Setup[*ShiftExpression*] () propagates the call to Setup to every nonterminal in the expansion of *ShiftExpression*.

Evaluation

```
proc Eval[ShiftExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [ShiftExpression ⇒ AdditiveExpression] do
      return Eval[AdditiveExpression](env, phase);
   [ShiftExpression₀ ⇒ ShiftExpression₁ << AdditiveExpression] do
      a: OBJECT ← readReference(Eval[ShiftExpression₁](env, phase), phase);
      b: OBJECT ← readReference(Eval[AdditiveExpression](env, phase), phase);
      return shiftLeft(a, b, phase);
   [ShiftExpression₀ ⇒ ShiftExpression₁ >> AdditiveExpression] do
      a: OBJECT ← readReference(Eval[ShiftExpression₁](env, phase), phase);
      b: OBJECT ← readReference(Eval[AdditiveExpression](env, phase), phase);
      return shiftRight(a, b, phase);
   [ShiftExpression₀ ⇒ ShiftExpression₁ >>> AdditiveExpression] do
      a: OBJECT ← readReference(Eval[ShiftExpression₁](env, phase), phase);
      b: OBJECT ← readReference(Eval[AdditiveExpression](env, phase), 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
    Float32 ∪ 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
    Float32 ∪ 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
        FLOAT32 □ 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$^{\text{allowIn}}$ □
  ShiftExpression
| *RelationalExpression*$^{\text{allowIn}}$ **<** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **>** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **<=** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **>=** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **is** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **as** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **in** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **instanceof** *ShiftExpression*

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

Validation

Validate[*RelationalExpression*$^{□}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *RelationalExpression*$^{□}$.

Setup

Setup[*RelationalExpression*$^{□}$] () propagates the call to Setup to every nonterminal in the expansion of *RelationalExpression*$^{□}$.

Evaluation

```
proc Eval[RelationalExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [RelationalExpression□ □ ShiftExpression] do
        return Eval[ShiftExpression](env, phase);
    [RelationalExpression□₀ □ RelationalExpression□₁ < ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        return isLess(a, b, phase);
    [RelationalExpression□₀ □ RelationalExpression□₁ > ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        return isLess(b, a, phase);
    [RelationalExpression□₀ □ RelationalExpression□₁ <= ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        return isLessOrEqual(a, b, phase);
    [RelationalExpression□₀ □ RelationalExpression□₁ >= ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        return isLessOrEqual(b, a, phase);
    [RelationalExpression□ □ RelationalExpression□ is ShiftExpression] do ????;
    [RelationalExpression□₀ □ RelationalExpression□₁ as ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        c: CLASS □ toClass(b);
        return c.implicitCoerce(a, true);
    [RelationalExpression^allowIn₀ □ RelationalExpression^allowIn₁ in ShiftExpression] do
        a: OBJECT □ readReference(Eval[RelationalExpression^allowIn₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[ShiftExpression](env, phase), phase);
        name: STRING □ toString(a, phase);
        qname: QUALIFIEDNAME □ QUALIFIEDNAME□namespace: publicNamespace, id: name□;
        c: CLASS □ objectType(b);
        return findVTableIndex(c, {qname}, read) ≠ none or findVTableIndex(c, {qname}, write) ≠ none or
                findCommonMember(b, {qname}, read, false) ≠ none or
                findCommonMember(b, {qname}, write, false) ≠ none;
    [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^β ⇒
      RelationalExpression^β
   |  EqualityExpression^β == RelationalExpression^β
   |  EqualityExpression^β != RelationalExpression^β
   |  EqualityExpression^β === RelationalExpression^β
   |  EqualityExpression^β !== RelationalExpression^β
```

Validation

Validate[*EqualityExpression*^β] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *EqualityExpression*^β.

Setup

Setup[*EqualityExpression*^β] () propagates the call to Setup to every nonterminal in the expansion of *EqualityExpression*^β.

Evaluation

```
proc Eval[EqualityExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [EqualityExpression^β ⇒ RelationalExpression^β] do
        return Eval[RelationalExpression^β](env, phase);
    [EqualityExpression^β_0 ⇒ EqualityExpression^β_1 == RelationalExpression^β] do
        a: OBJECT ← readReference(Eval[EqualityExpression^β_1](env, phase), phase);
        b: OBJECT ← readReference(Eval[RelationalExpression^β](env, phase), phase);
        return isEqual(a, b, phase);
    [EqualityExpression^β_0 ⇒ EqualityExpression^β_1 != RelationalExpression^β] do
        a: OBJECT ← readReference(Eval[EqualityExpression^β_1](env, phase), phase);
        b: OBJECT ← readReference(Eval[RelationalExpression^β](env, phase), phase);
        return not isEqual(a, b, phase);
    [EqualityExpression^β_0 ⇒ EqualityExpression^β_1 === RelationalExpression^β] do
        a: OBJECT ← readReference(Eval[EqualityExpression^β_1](env, phase), phase);
        b: OBJECT ← readReference(Eval[RelationalExpression^β](env, phase), phase);
        return isStrictlyEqual(a, b, phase);
    [EqualityExpression^β_0 ⇒ EqualityExpression^β_1 !== RelationalExpression^β] do
        a: OBJECT ← readReference(Eval[EqualityExpression^β_1](env, phase), phase);
        b: OBJECT ← readReference(Eval[RelationalExpression^β](env, phase), phase);
        return not isStrictlyEqual(a, b, phase)
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 □ SIMPLEINSTANCE □ DATE □ REGEXP □
            PACKAGE □ GLOBALOBJECT do
         case b of
            UNDEFINED □ NULL do return false;
            NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ METHODCLOSURE □ SIMPLEINSTANCE □ DATE □
                  REGEXP □ PACKAGE □ GLOBALOBJECT 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 □ GENERALNUMBER and b □ GENERALNUMBER then
      return generalNumberCompare(a, b) = equal
   else return a = b
   end if
end proc;
```

12.16 Binary Bitwise Operators

Syntax

```
BitwiseAndExpression□ □
      EqualityExpression□
   |  BitwiseAndExpression□ & EqualityExpression□

BitwiseXorExpression□ □
      BitwiseAndExpression□
   |  BitwiseXorExpression□ ^ BitwiseAndExpression□
```

```
BitwiseOrExpression□ □
      BitwiseXorExpression□
   |  BitwiseOrExpression□ | BitwiseXorExpression□
```

Validation

Validate[*BitwiseAndExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *BitwiseAndExpression*□.

Validate[*BitwiseXorExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *BitwiseXorExpression*□.

Validate[*BitwiseOrExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *BitwiseOrExpression*□.

Setup

Setup[*BitwiseAndExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *BitwiseAndExpression*□.

Setup[*BitwiseXorExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *BitwiseXorExpression*□.

Setup[*BitwiseOrExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *BitwiseOrExpression*□.

Evaluation

```
proc Eval[BitwiseAndExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [BitwiseAndExpression□ □ EqualityExpression□] do
      return Eval[EqualityExpression□](env, phase);
   [BitwiseAndExpression□_0 □ BitwiseAndExpression□_1 & EqualityExpression□] do
      a: OBJECT □ readReference(Eval[BitwiseAndExpression□_1](env, phase), phase);
      b: OBJECT □ readReference(Eval[EqualityExpression□](env, phase), 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
      a: OBJECT □ readReference(Eval[BitwiseXorExpression□_1](env, phase), phase);
      b: OBJECT □ readReference(Eval[BitwiseAndExpression□](env, phase), 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
      a: OBJECT □ readReference(Eval[BitwiseOrExpression□_1](env, phase), phase);
      b: OBJECT □ readReference(Eval[BitwiseXorExpression□](env, phase), 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: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(x));
        j: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(y));
        k: $\{-2^{63} \ldots 2^{63} - 1\}$ □ bitwiseAnd(i, j);
        if x □ ULONG or y □ ULONG then return ULONG□value: unsignedWrap64(k)□
        else return LONG□value: k□
        end if
    else
        i: $\{-2^{31} \ldots 2^{31} - 1\}$ □ signedWrap32(truncateToInteger(x));
        j: $\{-2^{31} \ldots 2^{31} - 1\}$ □ 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: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(x));
        j: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(y));
        k: $\{-2^{63} \ldots 2^{63} - 1\}$ □ bitwiseXor(i, j);
        if x □ ULONG or y □ ULONG then return ULONG□value: unsignedWrap64(k)□
        else return LONG□value: k□
        end if
    else
        i: $\{-2^{31} \ldots 2^{31} - 1\}$ □ signedWrap32(truncateToInteger(x));
        j: $\{-2^{31} \ldots 2^{31} - 1\}$ □ 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: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(x));
        j: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(truncateToInteger(y));
        k: $\{-2^{63} \ldots 2^{63} - 1\}$ □ bitwiseOr(i, j);
        if x □ ULONG or y □ ULONG then return ULONG□value: unsignedWrap64(k)□
        else return LONG□value: k□
        end if
    else
        i: $\{-2^{31} \ldots 2^{31} - 1\}$ □ signedWrap32(truncateToInteger(x));
        j: $\{-2^{31} \ldots 2^{31} - 1\}$ □ 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

Validate[*LogicalAndExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *LogicalAndExpression*□.

Validate[*LogicalXorExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *LogicalXorExpression*□.

Validate[*LogicalOrExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *LogicalOrExpression*□.

Setup

Setup[*LogicalAndExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *LogicalAndExpression*□.

Setup[*LogicalXorExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *LogicalXorExpression*□.

Setup[*LogicalOrExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *LogicalOrExpression*□.

Evaluation

```
proc Eval[LogicalAndExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [LogicalAndExpression□ □ BitwiseOrExpression□] do
    return Eval[BitwiseOrExpression□](env, phase);
  [LogicalAndExpression□_0 □ LogicalAndExpression□_1 && BitwiseOrExpression□] do
    a: OBJECT □ readReference(Eval[LogicalAndExpression□_1](env, phase), phase);
    if toBoolean(a, phase) then
      return readReference(Eval[BitwiseOrExpression□](env, phase), 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
        a: OBJECT □ readReference(Eval[LogicalXorExpression□₁](env, phase), phase);
        b: OBJECT □ readReference(Eval[LogicalAndExpression□](env, phase), 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
        a: OBJECT □ readReference(Eval[LogicalOrExpression□₁](env, phase), phase);
        if toBoolean(a, phase) then return a
        else return readReference(Eval[LogicalXorExpression□](env, phase), phase)
        end if
end proc;
```

12.18 Conditional Operator

Syntax

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

NonAssignmentExpression□ □
    LogicalOrExpression□
  | LogicalOrExpression□ ? NonAssignmentExpression□ : NonAssignmentExpression□
```

Validation

Validate[*ConditionalExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ConditionalExpression*□.

Validate[*NonAssignmentExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *NonAssignmentExpression*□.

Setup

Setup[*ConditionalExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *ConditionalExpression*□.

Setup[*NonAssignmentExpression*□] () propagates the call to Setup to every nonterminal in the expansion of *NonAssignmentExpression*□.

Evaluation

```
proc Eval[ConditionalExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ConditionalExpression□ □ LogicalOrExpression□] do
        return Eval[LogicalOrExpression□](env, phase);
    [ConditionalExpression□ □ LogicalOrExpression□ ? AssignmentExpression□1 : AssignmentExpression□2] do
        a: OBJECT □ readReference(Eval[LogicalOrExpression□](env, phase), phase);
        if toBoolean(a, phase) then
            return readReference(Eval[AssignmentExpression□1](env, phase), phase)
        else return readReference(Eval[AssignmentExpression□2](env, phase), 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
        a: OBJECT □ readReference(Eval[LogicalOrExpression□](env, phase), phase);
        if toBoolean(a, phase) then
            return readReference(Eval[NonAssignmentExpression□1](env, phase), phase)
        else return readReference(Eval[NonAssignmentExpression□2](env, phase), 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$^{\beta}$] (cxt: CONTEXT, env: ENVIRONMENT)
    [AssignmentExpression$^{\beta}$ ⇒ ConditionalExpression$^{\beta}$] do
        Validate[ConditionalExpression$^{\beta}$](cxt, env);
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression = AssignmentExpression$^{\beta}_{1}$] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression$^{\beta}_{1}$](cxt, env);
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression CompoundAssignment AssignmentExpression$^{\beta}_{1}$] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression$^{\beta}_{1}$](cxt, env);
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression LogicalAssignment AssignmentExpression$^{\beta}_{1}$] do
        Validate[PostfixExpression](cxt, env);
        Validate[AssignmentExpression$^{\beta}_{1}$](cxt, env)
end proc;
```

Setup

```
proc Setup[AssignmentExpression$^{\beta}$] ()
    [AssignmentExpression$^{\beta}$ ⇒ ConditionalExpression$^{\beta}$] do Setup[ConditionalExpression$^{\beta}$]();
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression = AssignmentExpression$^{\beta}_{1}$] do
        Setup[PostfixExpression]();
        Setup[AssignmentExpression$^{\beta}_{1}$]();
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression CompoundAssignment AssignmentExpression$^{\beta}_{1}$] do
        Setup[PostfixExpression]();
        Setup[AssignmentExpression$^{\beta}_{1}$]();
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression LogicalAssignment AssignmentExpression$^{\beta}_{1}$] do
        Setup[PostfixExpression]();
        Setup[AssignmentExpression$^{\beta}_{1}$]()
end proc;
```

Evaluation

```
proc Eval[AssignmentExpression$^{\beta}$] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [AssignmentExpression$^{\beta}$ ⇒ ConditionalExpression$^{\beta}$] do
        return Eval[ConditionalExpression$^{\beta}$](env, phase);
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression = AssignmentExpression$^{\beta}_{1}$] do
        if phase = compile then throw compileExpressionError end if;
        ra: OBJORREF ← Eval[PostfixExpression](env, phase);
        b: OBJECT ← readReference(Eval[AssignmentExpression$^{\beta}_{1}$](env, phase), phase);
        writeReference(ra, b, phase);
        return b;
    [AssignmentExpression$^{\beta}_{0}$ ⇒ PostfixExpression CompoundAssignment AssignmentExpression$^{\beta}_{1}$] do
        if phase = compile then throw compileExpressionError end if;
        rLeft: OBJORREF ← Eval[PostfixExpression](env, phase);
        oLeft: OBJECT ← readReference(rLeft, phase);
        oRight: OBJECT ← readReference(Eval[AssignmentExpression$^{\beta}_{1}$](env, phase), 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

Validate[*ListExpression*^{β}] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *ListExpression*^{β}.

Setup

Setup[*ListExpression*^{β}] () propagates the call to Setup to every nonterminal in the expansion of *ListExpression*^{β}.

Evaluation

```
proc Eval[ListExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [ListExpression^β ⇒ AssignmentExpression^β] do
      return Eval[AssignmentExpression^β](env, phase);
   [ListExpression^β_0 ⇒ ListExpression^β_1 , AssignmentExpression^β] do
      readReference(Eval[ListExpression^β_1](env, phase), phase);
      return readReference(Eval[AssignmentExpression^β](env, phase), phase)
end proc;
```

```
proc EvalAsList[ListExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
   [ListExpression^β ⇒ AssignmentExpression^β] do
      elt: OBJECT ← readReference(Eval[AssignmentExpression^β](env, phase), phase);
      return [elt];
   [ListExpression^β_0 ⇒ ListExpression^β_1 , AssignmentExpression^β] do
      elts: OBJECT[] ← EvalAsList[ListExpression^β_1](env, phase);
      elt: OBJECT ← readReference(Eval[AssignmentExpression^β](env, phase), phase);
      return elts ⊕ [elt]
end proc;
```

12.21 Type Expressions

Syntax

TypeExpression^β ⇒ *NonAssignmentExpression*^{β}

Validation

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

Setup and Evaluation

```
proc SetupAndEval[TypeExpression^β ⇒ NonAssignmentExpression^β] (env: ENVIRONMENT): CLASS
   Setup[NonAssignmentExpression^β]();
   o: OBJECT ← readReference(Eval[NonAssignmentExpression^β](env, compile), compile);
   return toClass(o)
end proc;
```

13 Statements

Syntax

ω ∈ {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 Validate[SwitchStatement](cxt, env, jt);
    [Statement□ □ DoStatement Semicolon□] do Validate[DoStatement](cxt, env, sl, jt);
    [Statement□ □ WhileStatement□] do Validate[WhileStatement□](cxt, env, sl, jt);
    [Statement□ □ ForStatement□] do Validate[ForStatement□](cxt, env, sl, jt);
    [Statement□ □ WithStatement□] do Validate[WithStatement□](cxt, env, jt);
    [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 Validate[TryStatement](cxt, env, jt)
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);
        Setup[Attributes]();
        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;
```

Setup

Setup[*Statement*□] () propagates the call to Setup to every nonterminal in the expansion of *Statement*□.

```
proc Setup[Substatement□] ()
    [Substatement□ □ EmptyStatement] do nothing;
    [Substatement□ □ Statement□] do Setup[Statement□]();
    [Substatement□ □ SimpleVariableDefinition Semicolon□] do
        Setup[SimpleVariableDefinition]();
    [Substatement□ □ Attributes [no line break] { Substatements }] do
        if Enabled[Substatement□] then Setup[Substatements]() end if
end proc;
```

Setup[*Substatements*] () propagates the call to Setup to every nonterminal in the expansion of *Substatements*.

Setup[*SubstatementsPrefix*] () propagates the call to Setup to every nonterminal in the expansion of *SubstatementsPrefix*.

```
proc Setup[Semicolon□] ()
    [Semicolon□ □ ;] do nothing;
    [Semicolon□ □ VirtualSemicolon] do nothing;
    [Semicolon^abbrev □ «empty»] do nothing;
    [Semicolon^noShortIf □ «empty»] do nothing
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 return Eval[SwitchStatement](env, d);
    [Statement□ □ DoStatement Semicolon□] do return Eval[DoStatement](env, d);
    [Statement□ □ WhileStatement□] do return Eval[WhileStatement□](env, d);
    [Statement□ □ ForStatement□] do return Eval[ForStatement□](env, d);
    [Statement□ □ WithStatement□] do return Eval[WithStatement□](env, d);
    [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 return Eval[TryStatement](env, d)
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^ω ⇒ Attributes [no line break] { Substatements }] do
        if Enabled[Substatement^ω] 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;
```

Setup

```
proc Setup[ExpressionStatement ⇒ [lookahead∉{function, {}] ListExpression^allowIn] ()
    Setup[ListExpression^allowIn]()
end proc;
```

Evaluation

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

13.3 Super Statement

Syntax

```
SuperStatement → super Arguments
```

Validation

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

Setup

```
proc Setup[SuperStatement → super Arguments] ()
    Setup[Arguments]()
end proc;
```

Evaluation

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

13.4 Block Statement

Syntax

```
Block → { Directives }
```

Validation

```
CompileFrame[Block]: BLOCKFRAME;

proc Validate[Block → { Directives }] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY)
    compileFrame: BLOCKFRAME ← new BLOCKFRAME⟨⟨localBindings: {}, 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;
```

Setup

```
proc Setup[Block → { Directives }] ()
    Setup[Directives]()
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 □ instantiateBlockFrame(compileFrame, 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;
```

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;
```

Setup

```
proc Setup[LabeledStatement□ □ Identifier : Substatement□] ()
    Setup[Substatement□]()
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;
```

Setup

Setup[*IfStatement*$^{□}$] () propagates the call to Setup to every nonterminal in the expansion of *IfStatement*$^{□}$.

Evaluation

```
proc Eval[IfStatement$^{□}$] (env: ENVIRONMENT, d: OBJECT): OBJECT
   [IfStatement$^{abbrev}$ □ if ParenListExpression Substatement$^{abbrev}$] do
      o: OBJECT □ readReference(Eval[ParenListExpression](env, run), run);
      if toBoolean(o, run) then return Eval[Substatement$^{abbrev}$](env, d)
      else return d
      end if;
   [IfStatement$^{full}$ □ if ParenListExpression Substatement$^{full}$] do
      o: OBJECT □ readReference(Eval[ParenListExpression](env, run), 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
      o: OBJECT □ readReference(Eval[ParenListExpression](env, run), 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

Semantics

```
tuple SWITCHKEY
   key: OBJECT
end tuple;

SWITCHGUARD = SWITCHKEY ∪ {default} ∪ OBJECT;
```

Syntax

```
SwitchStatement ⇒ switch ParenListExpression { CaseElements }

CaseElements ⇒
      «empty»
   |  CaseLabel
   |  CaseLabel CaseElementsPrefix CaseElement^abbrev

CaseElementsPrefix ⇒
      «empty»
   |  CaseElementsPrefix CaseElement^full

CaseElement^ω ⇒
      Directive^ω
   |  CaseLabel

CaseLabel ⇒
      case ListExpression^allowIn :
   |  default :
```

Validation

```
CompileFrame[SwitchStatement]: BLOCKFRAME;

proc Validate[SwitchStatement ⇒ switch ParenListExpression { CaseElements }]
      (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   if NDefaults[CaseElements] > 1 then throw syntaxError end if;
   Validate[ParenListExpression](cxt, env);
   jt2: JUMPTARGETS ← JUMPTARGETS⟨⟨breakTargets: jt.breakTargets ∪ {default},
         continueTargets: jt.continueTargets⟩⟩;
   compileFrame: BLOCKFRAME ← new BLOCKFRAME⟨⟨localBindings: {}, plurality: plural⟩⟩;
   CompileFrame[SwitchStatement] ← compileFrame;
   Validate[CaseElements](cxt, [compileFrame] ⊕ env, jt2)
end proc;

NDefaults[CaseElements]: INTEGER;
   NDefaults[CaseElements ⇒ «empty»] = 0;
   NDefaults[CaseElements ⇒ CaseLabel] = NDefaults[CaseLabel];
   NDefaults[CaseElements ⇒ CaseLabel CaseElementsPrefix CaseElement^abbrev]
         = NDefaults[CaseLabel] + NDefaults[CaseElementsPrefix] + NDefaults[CaseElement^abbrev];
```

```
proc Validate[CaseElements] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS): CONTEXT
    [CaseElements ⇒ «empty»] do return cxt;
    [CaseElements ⇒ CaseLabel] do Validate[CaseLabel](cxt, env); return cxt;
    [CaseElements ⇒ CaseLabel CaseElementsPrefix CaseElement$^{\text{abbrev}}$] do
        Validate[CaseLabel](cxt, env);
        cxt2: CONTEXT ← Validate[CaseElementsPrefix](cxt, env, jt);
        return Validate[CaseElement$^{\text{abbrev}}$](cxt2, env, jt)
end proc;

NDefaults[CaseElementsPrefix]: INTEGER;
    NDefaults[CaseElementsPrefix ⇒ «empty»] = 0;
    NDefaults[CaseElementsPrefix$_0$ ⇒ CaseElementsPrefix$_1$ CaseElement$^{\text{full}}$]
            = NDefaults[CaseElementsPrefix$_1$] + NDefaults[CaseElement$^{\text{full}}$];

proc Validate[CaseElementsPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS): CONTEXT
    [CaseElementsPrefix ⇒ «empty»] do return cxt;
    [CaseElementsPrefix$_0$ ⇒ CaseElementsPrefix$_1$ CaseElement$^{\text{full}}$] do
        cxt2: CONTEXT ← Validate[CaseElementsPrefix$_1$](cxt, env, jt);
        return Validate[CaseElement$^{\text{full}}$](cxt2, env, jt)
end proc;

NDefaults[CaseElement$^{\omega}$]: INTEGER;
    NDefaults[CaseElement$^{\omega}$ ⇒ Directive$^{\omega}$] = 0;
    NDefaults[CaseElement$^{\omega}$ ⇒ CaseLabel] = NDefaults[CaseLabel];

proc Validate[CaseElement$^{\omega}$] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS): CONTEXT
    [CaseElement$^{\omega}$ ⇒ Directive$^{\omega}$] do
        return Validate[Directive$^{\omega}$](cxt, env, jt, plural, none);
    [CaseElement$^{\omega}$ ⇒ CaseLabel] do Validate[CaseLabel](cxt, env); return cxt
end proc;

NDefaults[CaseLabel]: INTEGER;
    NDefaults[CaseLabel ⇒ case ListExpression$^{\text{allowIn}}$ :] = 0;
    NDefaults[CaseLabel ⇒ default :] = 1;
```

Validate[*CaseLabel*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *CaseLabel*.

Setup

Setup[*SwitchStatement*] () propagates the call to Setup to every nonterminal in the expansion of *SwitchStatement*.

Setup[*CaseElements*] () propagates the call to Setup to every nonterminal in the expansion of *CaseElements*.

Setup[*CaseElementsPrefix*] () propagates the call to Setup to every nonterminal in the expansion of *CaseElementsPrefix*.

Setup[*CaseElement*$^{\omega}$] () propagates the call to Setup to every nonterminal in the expansion of *CaseElement*$^{\omega}$.

Setup[*CaseLabel*] () propagates the call to Setup to every nonterminal in the expansion of *CaseLabel*.

Evaluation

```
proc Eval[SwitchStatement → switch ParenListExpression { CaseElements }]
      (env: Environment, d: Object): Object
   key: Object ← readReference(Eval[ParenListExpression](env, run), run);
   compileFrame: BlockFrame ← CompileFrame[SwitchStatement];
   runtimeFrame: BlockFrame ← instantiateBlockFrame(compileFrame, env);
   runtimeEnv: Environment ← [runtimeFrame] ⊕ env;
   result: SwitchGuard ← Eval[CaseElements](runtimeEnv, SwitchKey⟨key: key⟩, d);
   if result ∈ Object then return result end if;
   note result = SwitchKey⟨key: key⟩;
   result ← Eval[CaseElements](runtimeEnv, default, d);
   if result ∈ Object then return result end if;
   note result = default;
   return d
end proc;

proc Eval[CaseElements] (env: Environment, guard: SwitchGuard, d: Object): SwitchGuard
   [CaseElements → «empty»] do return guard;
   [CaseElements → CaseLabel] do return Eval[CaseLabel](env, guard, d);
   [CaseElements → CaseLabel CaseElementsPrefix CaseElement^abbrev] do
      guard2: SwitchGuard ← Eval[CaseLabel](env, guard, d);
      guard3: SwitchGuard ← Eval[CaseElementsPrefix](env, guard2, d);
      return Eval[CaseElement^abbrev](env, guard3, d)
end proc;

proc Eval[CaseElementsPrefix] (env: Environment, guard: SwitchGuard, d: Object): SwitchGuard
   [CaseElementsPrefix → «empty»] do return guard;
   [CaseElementsPrefix_0 → CaseElementsPrefix_1 CaseElement^full] do
      guard2: SwitchGuard ← Eval[CaseElementsPrefix_1](env, guard, d);
      return Eval[CaseElement^full](env, guard2, d)
end proc;

proc Eval[CaseElement^ω] (env: Environment, guard: SwitchGuard, d: Object): SwitchGuard
   [CaseElement^ω → Directive^ω] do
      case guard of
         SwitchKey ∪ {default} do return guard;
         Object do return Eval[Directive^ω](env, guard)
      end case;
   [CaseElement^ω → CaseLabel] do return Eval[CaseLabel](env, guard, d)
end proc;

proc Eval[CaseLabel] (env: Environment, guard: SwitchGuard, d: Object): SwitchGuard
   [CaseLabel → case ListExpression^allowIn :] do
      case guard of
         {default} ∪ Object do return guard;
         SwitchKey do
            label: Object ← readReference(Eval[ListExpression^allowIn](env, run), run);
            if isStrictlyEqual(guard.key, label, run) then return d
            else return guard
            end if
      end case;
```

```
    [CaseLabel □ default :] do
       case guard of
          SWITCHKEY □ OBJECT do return guard;
          {default} do return d
       end case
  end proc;
```

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;
```

Setup

Setup[*DoStatement*] () propagates the call to Setup to every nonterminal in the expansion of *DoStatement*.

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;
           o: OBJECT □ readReference(Eval[ParenListExpression](env, run), 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□ □ while ParenListExpression Substatement□
```

Validation

```
Labels[WhileStatement□]: LABEL{};

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

Setup

Setup[*WhileStatement*□] () propagates the call to Setup to every nonterminal in the expansion of *WhileStatement*□.

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$^{\text{noIn}}$
   |  VariableDefinitionKind VariableBindingList$^{\text{noIn}}$
   |  Attributes [no line break] VariableDefinitionKind VariableBindingList$^{\text{noIn}}$

ForInBinding □
      PostfixExpression
   |  VariableDefinitionKind VariableBinding$^{\text{noIn}}$
   |  Attributes [no line break] VariableDefinitionKind VariableBinding$^{\text{noIn}}$
```

Validation

```
proc Validate[ForStatement□] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
   [ForStatement□ □ for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□] do
      ????;
   [ForStatement□ □ for ( ForInBinding in ListExpression$^{\text{allowIn}}$ ) Substatement□] do
      ????
end proc;
```

Setup

```
proc Setup[ForStatement□] ()
   [ForStatement□ □ for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□] do
      ????;
   [ForStatement□ □ for ( ForInBinding in ListExpression$^{\text{allowIn}}$ ) Substatement□] do
      ????
end proc;
```

Evaluation

```
proc Eval[ForStatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
   [ForStatement□ □ for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□] do
      ????;
   [ForStatement□ □ for ( ForInBinding in ListExpression$^{\text{allowIn}}$ ) Substatement□] do
      ????
end proc;
```

13.11 With Statement

Syntax

```
WithStatement□ □ with ParenListExpression Substatement□
```

Validation

```
proc Validate[WithStatement□ □ with ParenListExpression Substatement□]
      (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   Validate[ParenListExpression](cxt, env);
   Validate[Substatement□](cxt, env, {}, jt)
end proc;
```

Setup

Setup[*WithStatement*□] () propagates the call to Setup to every nonterminal in the expansion of *WithStatement*□.

Evaluation

```
proc Eval[WithStatement^ω ⇒ with ParenListExpression Substatement^ω] (env: Environment, d: Object): Object
  ????
end proc;
```

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;
```

Setup

```
proc Setup[ContinueStatement] ()
  [ContinueStatement ⇒ continue] do nothing;
  [ContinueStatement ⇒ continue [no line break] Identifier] do nothing
end proc;

proc Setup[BreakStatement] ()
  [BreakStatement ⇒ break] do nothing;
  [BreakStatement ⇒ break [no line break] Identifier] do nothing
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;
```

Setup

Setup[*ReturnStatement*] () propagates the call to Setup to every nonterminal in the expansion of *ReturnStatement*.

Evaluation

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

13.14 Throw Statement

Syntax

```
ThrowStatement ⇒ throw [no line break] ListExpression^allowIn
```

Validation

```
proc Validate[ThrowStatement ⇒ throw [no line break] ListExpression^allowIn] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpression^allowIn](cxt, env)
end proc;
```

Setup

```
proc Setup[ThrowStatement ⇒ throw [no line break] ListExpression^allowIn] ()
  Setup[ListExpression^allowIn]()
end proc;
```

Evaluation

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

13.15 Try Statement

Syntax

```
TryStatement ⇒
      try Block CatchClauses
   |  try Block CatchClausesOpt finally Block

CatchClausesOpt ⇒
      «empty»
   |  CatchClauses

CatchClauses ⇒
      CatchClause
   |  CatchClauses CatchClause

CatchClause ⇒ catch ( Parameter ) Block
```

Validation

```
proc Validate[TryStatement] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   [TryStatement ⇒ try Block CatchClauses] do
      Validate[Block](cxt, env, jt, plural);
      Validate[CatchClauses](cxt, env, jt);
   [TryStatement ⇒ try Block_1 CatchClausesOpt finally Block_2] do
      Validate[Block_1](cxt, env, jt, plural);
      Validate[CatchClausesOpt](cxt, env, jt);
      Validate[Block_2](cxt, env, jt, plural)
end proc;
```

Validate[*CatchClausesOpt*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS) propagates the call to Validate to every nonterminal in the expansion of *CatchClausesOpt*.

Validate[*CatchClauses*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS) propagates the call to Validate to every nonterminal in the expansion of *CatchClauses*.

```
proc Validate[CatchClause ⇒ catch ( Parameter ) Block] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   ????
end proc;
```

Setup

Setup[*TryStatement*] () propagates the call to Setup to every nonterminal in the expansion of *TryStatement*.

Setup[*CatchClausesOpt*] () propagates the call to Setup to every nonterminal in the expansion of *CatchClausesOpt*.

Setup[*CatchClauses*] () propagates the call to Setup to every nonterminal in the expansion of *CatchClauses*.

```
proc Setup[CatchClause ⇒ catch ( Parameter ) Block] ()
   ????
end proc;
```

Evaluation

```
proc Eval[TryStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [TryStatement  try Block CatchClauses] do
        try return Eval[Block](env, d)
        catch x: SEMANTICEXCEPTION do
            if x  THROWNVALUE then throw x end if;
            exception: OBJECT  x.value;
            r: OBJECT  {reject}  Eval[CatchClauses](env, exception);
            if r ≠ reject then return r else throw x end if
        end try;
    [TryStatement  try Block$_1$ CatchClausesOpt finally Block$_2$] do
        result: OBJECT  SEMANTICEXCEPTION;
        try result  Eval[Block$_1$](env, d)
        catch x: SEMANTICEXCEPTION do result  x
        end try;
        if result  THROWNVALUE then
            exception: OBJECT  result.value;
            try
                r: OBJECT  {reject}  Eval[CatchClausesOpt](env, exception);
                if r ≠ reject then result  r end if
            catch y: SEMANTICEXCEPTION do result  y
            end try
        end if;
        Eval[Block$_2$](env, undefined);
        case result of
            OBJECT do return result;
            SEMANTICEXCEPTION do throw result
        end case
end proc;

proc Eval[CatchClausesOpt] (env: ENVIRONMENT, exception: OBJECT): OBJECT  {reject}
    [CatchClausesOpt  «empty»] do return reject;
    [CatchClausesOpt  CatchClauses] do return Eval[CatchClauses](env, exception)
end proc;

proc Eval[CatchClauses] (env: ENVIRONMENT, exception: OBJECT): OBJECT  {reject}
    [CatchClauses  CatchClause] do return Eval[CatchClause](env, exception);
    [CatchClauses$_0$  CatchClauses$_1$ CatchClause] do
        r: OBJECT  {reject}  Eval[CatchClauses$_1$](env, exception);
        if r ≠ reject then return r else return Eval[CatchClause](env, exception) end if
end proc;

proc Eval[CatchClause  catch ( Parameter ) Block] (env: ENVIRONMENT, exception: OBJECT): OBJECT  {reject}
    ????
end proc;
```

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

proc Validate[AnnotatableDirective$^{\omega}$]
        (cxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
    [AnnotatableDirective$^{\omega}$ → ExportDefinition Semicolon$^{\omega}$] do ????;
    [AnnotatableDirective$^{\omega}$ → VariableDefinition Semicolon$^{\omega}$] do
        Validate[VariableDefinition](cxt, env, attr);
        return cxt;
    [AnnotatableDirective$^{\omega}$ → FunctionDefinition] do
        Validate[FunctionDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective$^{\omega}$ → ClassDefinition] do
        Validate[ClassDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective$^{\omega}$ → NamespaceDefinition Semicolon$^{\omega}$] do
        Validate[NamespaceDefinition](cxt, env, pl, attr);
        return cxt;
    [AnnotatableDirective$^{\omega}$ → ImportDirective Semicolon$^{\omega}$] do ????;
    [AnnotatableDirective$^{\omega}$ → UseDirective Semicolon$^{\omega}$] 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;
```

Setup

```
proc Setup[Directive□] ()
    [Directive□ □ EmptyStatement] do nothing;
    [Directive□ □ Statement□] do Setup[Statement□]();
    [Directive□ □ AnnotatableDirective□] do Setup[AnnotatableDirective□]();
    [Directive□ □ Attributes [no line break] AnnotatableDirective□] do
        if Enabled[Directive□] then Setup[AnnotatableDirective□]() end if;
    [Directive□ □ Attributes [no line break] { Directives }] do
        if Enabled[Directive□] then Setup[Directives]() end if;
    [Directive□ □ PackageDefinition] do ????;
    [Directive□ □ Pragma Semicolon□] do nothing
end proc;

proc Setup[AnnotatableDirective□] ()
    [AnnotatableDirective□ □ ExportDefinition Semicolon□] do ????;
    [AnnotatableDirective□ □ VariableDefinition Semicolon□] do
        Setup[VariableDefinition]();
    [AnnotatableDirective□ □ FunctionDefinition] do Setup[FunctionDefinition]();
    [AnnotatableDirective□ □ ClassDefinition] do Setup[ClassDefinition]();
    [AnnotatableDirective□ □ NamespaceDefinition Semicolon□] do nothing;
    [AnnotatableDirective□ □ ImportDirective Semicolon□] do ????;
    [AnnotatableDirective□ □ UseDirective Semicolon□] do nothing
end proc;
```

Setup[*Directives*] () propagates the call to Setup to every nonterminal in the expansion of *Directives*.

Setup[*DirectivesPrefix*] () propagates the call to Setup to every nonterminal in the expansion of *DirectivesPrefix*.

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^ω ⇒ FunctionDefinition] do return d;
    [AnnotatableDirective^ω ⇒ ClassDefinition] do return Eval[ClassDefinition](env, d);
    [AnnotatableDirective^ω ⇒ NamespaceDefinition Semicolon^ω] do return d;
    [AnnotatableDirective^ω ⇒ ImportDirective Semicolon^ω] do ????;
    [AnnotatableDirective^ω ⇒ UseDirective Semicolon^ω] 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^ω]: 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

Validate[*Attributes*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *Attributes*.

Validate[*AttributeCombination*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *AttributeCombination*.

Validate[*Attribute*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *Attribute*.

```
proc Validate[NonexpressionAttribute] (cxt: CONTEXT, env: ENVIRONMENT)
    [NonexpressionAttribute ⇒ final] do nothing;
    [NonexpressionAttribute ⇒ private] do
        if getEnclosingClass(env) = none then throw syntaxError end if;
    [NonexpressionAttribute ⇒ static] do nothing
end proc;
```

Setup

Setup[*Attributes*] () propagates the call to Setup to every nonterminal in the expansion of *Attributes*.

Setup[*AttributeCombination*] () propagates the call to Setup to every nonterminal in the expansion of *AttributeCombination*.

Setup[*Attribute*] () propagates the call to Setup to every nonterminal in the expansion of *Attribute*.

```
proc Setup[NonexpressionAttribute] ()
    [NonexpressionAttribute ⇒ final] do nothing;
    [NonexpressionAttribute ⇒ private] do nothing;
    [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
        a: OBJECT ← readReference(Eval[AttributeExpression](env, phase), 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 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);
    Setup[ParenListExpression]();
    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
   |  - NegatedMinLong
   |  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] = generalNumberNegate(Value[Number]);
    Value[PragmaArgument □ - NegatedMinLong] = LONG□value: –2^63□
    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_f64} then return cxt end if;
        if value □ {1.0_f64, 2.0_f64, 3.0_f64} 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□

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;

Validate[VariableBindingList□]
        (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN) propagates the call to
        Validate to every nonterminal in the expansion of VariableBindingList□.

CompileEnv[VariableBinding□]: ENVIRONMENT;

CompileVar[VariableBinding□]: DYNAMICVAR □ VARIABLE □ INSTANCEVARIABLE;

OverriddenIndices[VariableBinding□]: VTABLEINDEX{};

Multiname[VariableBinding□]: MULTINAME;
```

```
proc Validate[VariableBinding^β ⟶ TypedIdentifier^β VariableInitialisation^β]
        (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN)
    Validate[TypedIdentifier^β](cxt, env);
    Validate[VariableInitialisation^β](cxt, env);
    CompileEnv[VariableBinding^β] ← env;
    name: STRING ← Name[TypedIdentifier^β];
    if not cxt.strict and getRegionalFrame(env) ∈ GLOBALOBJECT ∪ PARAMETERFRAME and not immutable and
            attr = none and Plain[TypedIdentifier^β] then
        qname: QUALIFIEDNAME ← QUALIFIEDNAME⟨⟨namespace: publicNamespace, id: name⟩⟩;
        Multiname[VariableBinding^β] ← {qname};
        CompileVar[VariableBinding^β] ← defineHoistedVar(env, name, undefined)
    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;
        case memberMod of
            {none, static} do
                proc evalType(): CLASS
                    type: CLASSOPT ← SetupAndEval[TypedIdentifier^β](env);
                    if type = none then return objectClass end if;
                    return type
                end proc;
                proc evalInitialiser(): OBJECT
                    Setup[VariableInitialisation^β]();
                    value: OBJECTOPT ← Eval[VariableInitialisation^β](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: VARIABLE ← new VARIABLE⟨⟨type: evalType, value: initialValue, immutable: immutable⟩⟩;
                multiname: MULTINAME ← defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit,
                        readWrite, v);
                Multiname[VariableBinding^β] ← multiname;
                CompileVar[VariableBinding^β] ← v;
            {virtual, final} do
                c: CLASS ← env[0];
                proc evalInitialValue(): OBJECTOPT
                    return Eval[VariableInitialisation^β](env, run)
                end proc;
                v: INSTANCEVARIABLE ← new INSTANCEVARIABLE⟨⟨final: memberMod = final,
                        evalInitialValue: evalInitialValue, immutable: immutable⟩⟩;
                OverriddenIndices[VariableBinding^β] ← defineInstanceMember(c, cxt, name, a.namespaces,
                        a.overrideMod, a.explicit, readWrite, v);
                CompileVar[VariableBinding^β] ← v;
            {constructor} do throw definitionError
        end case
    end if
end proc;
```

Validate[*VariableInitialisation*^β] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *VariableInitialisation*^β.

Validate[*VariableInitialiser*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *VariableInitialiser*□.

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

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

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

Setup

```
proc Setup[VariableDefinition □ VariableDefinitionKind VariableBindingList^allowIn] ()
    Setup[VariableBindingList^allowIn]()
end proc;
```

Setup[*VariableBindingList*□] () propagates the call to Setup to every nonterminal in the expansion of *VariableBindingList*□.

```
proc Setup[VariableBinding^β ⇒ TypedIdentifier^β VariableInitialisation^β] ()
  env: ENVIRONMENT ← CompileEnv[VariableBinding^β];
  v: DYNAMICVAR ∪ VARIABLE ∪ INSTANCEVARIABLE ← CompileVar[VariableBinding^β];
  case v of
    DYNAMICVAR do Setup[VariableInitialisation^β]();
    VARIABLE do
      type: CLASS ← getVariableType(v, compile);
      case v.value of
        OBJECT do nothing;
        {inaccessible} do Setup[VariableInitialisation^β]();
        () → OBJECT do
          v.value ← inaccessible;
          Setup[VariableInitialisation^β]();
          try
            value: OBJECTOPT ← Eval[VariableInitialisation^β](env, compile);
            if value ≠ none then
              coercedValue: OBJECT ← type.implicitCoerce(value, false);
              v.value ← coercedValue
            end if
          catch x: SEMANTICEXCEPTION do
            if x ≠ compileExpressionError then throw x end if;
            note 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 case;
    INSTANCEVARIABLE do
      t: CLASSOPT ← SetupAndEval[TypedIdentifier^β](env);
      if t = none then
        c: CLASS ← env[0];
        overriddenTypes: CLASS{} ← {getInstanceMember(c.super, i).type |
            ∀i ∈ OverriddenIndices[VariableBinding^β]};
        if overriddenTypes = {} then t ← objectClass
        elsif |overriddenTypes| = 1 then t ← the one element of overriddenTypes
        else throw definitionError
        end if
      end if;
      v.type ← t;
      Setup[VariableInitialisation^β]()
  end case
end proc;
```

Setup[*VariableInitialisation*^β] () propagates the call to Setup to every nonterminal in the expansion of *VariableInitialisation*^β.

Setup[*VariableInitialiser*^β] () propagates the call to Setup to every nonterminal in the expansion of *VariableInitialiser*^β.

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 CompileVar[VariableBinding□] of
      DYNAMICVAR do
         value: OBJECTOPT □ Eval[VariableInitialisation□](env, run);
         if value ≠ none then
            lexicalWrite(env, Multiname[VariableBinding□], value, false, run)
         end if;
      VARIABLE do
         localFrame: FRAME □ env[0];
         members: LOCALMEMBER{} □ {b.content | □b □ localFrame.localBindings such that
               b.qname □ Multiname[VariableBinding□]};
         note 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 □ type.implicitCoerce(value, false)
            elsif immutable then coercedValue □ uninitialised
            else coercedValue □ type.defaultValue
            end if;
            v.value □ coercedValue
         end if;
      INSTANCEVARIABLE 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
      return readReference(Eval[AssignmentExpression□](env, phase), phase);
   [VariableInitialiser□ □ NonexpressionAttribute] do
      return Eval[NonexpressionAttribute](env, phase);
   [VariableInitialiser□ □ AttributeCombination] do
      return Eval[AttributeCombination](env, phase)
end proc;

proc SetupAndEval[TypedIdentifier□] (env: ENVIRONMENT): CLASSOPT
   [TypedIdentifier□ □ Identifier] do return none;
   [TypedIdentifier□ □ Identifier : TypeExpression□] do
      return SetupAndEval[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*$^{\omega}$ instead of a *Directive*$^{\omega}$ 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) ∉ GlobalObject ∪ ParameterFrame then
    throw syntaxError
  end if;
  Validate[UntypedVariableBindingList](cxt, env)
end proc;
```

Validate[*UntypedVariableBindingList*] (*cxt*: Context, *env*: Environment) propagates the call to Validate to every nonterminal in the expansion of *UntypedVariableBindingList*.

```
proc Validate[UntypedVariableBinding → Identifier VariableInitialisation$^{allowIn}$] (cxt: Context, env: Environment)
  Validate[VariableInitialisation$^{allowIn}$](cxt, env);
  defineHoistedVar(env, Name[Identifier], undefined)
end proc;
```

Setup

```
proc Setup[SimpleVariableDefinition → var UntypedVariableBindingList] ()
  Setup[UntypedVariableBindingList]()
end proc;
```

Setup[*UntypedVariableBindingList*] () propagates the call to Setup to every nonterminal in the expansion of *UntypedVariableBindingList*.

```
proc Setup[UntypedVariableBinding → Identifier VariableInitialisation$^{allowIn}$] ()
  Setup[VariableInitialisation$^{allowIn}$]()
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);
    [UntypedVariableBindingList_0 → UntypedVariableBindingList_1 , UntypedVariableBinding] do
        Eval[UntypedVariableBindingList_1](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 FunctionCommon*

FunctionName →
    Identifier
  | **`get`** [no line break] *Identifier*
  | **`set`** [no line break] *Identifier*

FunctionCommon → **`(`** *Parameters* **`)`** *Result Block*

Validation

EnclosingFrame[*FunctionDefinition*]: FRAME;

OverriddenIndices[*FunctionDefinition*]: VTABLEINDEX{};

```
proc Validate[FunctionDefinition ⇒ function FunctionName FunctionCommon]
      (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 Plain[FunctionCommon];
   prototype: Boolean ← unchecked or a.prototype;
   memberMod: MemberModifier ← a.memberMod;
   EnclosingFrame[FunctionDefinition] ← env[0];
   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;
   case memberMod of
      {none, static} do
         f: SimpleInstance ∪ UninstantiatedFunction;
         if kind ∈ {get, set} then ????
         else
            this: {none, inaccessible} ← prototype ? inaccessible : none;
            f ← ValidateStaticFunction[FunctionCommon](cxt, env, this, unchecked, prototype)
         end if;
         if pl = singular then f ← instantiateFunction(f, env) end if;
         if unchecked and attr = none and (env[0] ∈ GlobalObject or
               (env[0] ∈ BlockFrame and env[1] ∈ ParameterFrame)) then
            defineHoistedVar(env, name, f)
         else
            v: Variable ← new Variable⟨⟨type: functionClass, value: f, immutable: true⟩⟩;
            defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
         end if;
         OverriddenIndices[FunctionDefinition] ← {};
      {virtual, final} do
         note pl = singular;
         if kind ∈ {get, set} then ???? end if;
         Validate[FunctionCommon](cxt, env, inaccessible, false, prototype);
         method: InstanceMethod ← new InstanceMethod⟨⟨final: memberMod = final,
               signature: CompileFrame[FunctionCommon], call: EvalInstanceCall[FunctionCommon]⟩⟩;
         OverriddenIndices[FunctionDefinition] ← defineInstanceMember(env[0], cxt, name, a.namespaces,
               a.overrideMod, a.explicit, readWrite, method);
      {constructor} do
         note pl = singular;
         OverriddenIndices[FunctionDefinition] ← {};
         ????
   end case
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];

Plain[FunctionCommon → ( Parameters ) Result Block]: BOOLEAN = Plain[Parameters] and Plain[Result];

CompileEnv[FunctionCommon]: ENVIRONMENT;

CompileFrame[FunctionCommon]: PARAMETERFRAME;

proc Validate[FunctionCommon → ( Parameters ) Result Block] (cxt: CONTEXT, env: ENVIRONMENT,
      this: {none, inaccessible}, unchecked: BOOLEAN, prototype: BOOLEAN)
   compileFrame: PARAMETERFRAME ← new PARAMETERFRAME〈〈localBindings: {}, plurality: plural, this: this,
         unchecked: unchecked, prototype: prototype, positional: [], named: {}, rest: none,
         restAllowsNames: false〉〉;
   compileEnv: ENVIRONMENT ← [compileFrame] ⊕ env;
   CompileFrame[FunctionCommon] ← compileFrame;
   CompileEnv[FunctionCommon] ← compileEnv;
   Validate[Parameters](cxt, compileEnv, compileFrame);
   Validate[Result](cxt, compileEnv);
   Validate[Block](cxt, compileEnv, JUMPTARGETS〈〈breakTargets: {}, continueTargets: {}〉〉, plural)
end proc;

proc ValidateStaticFunction[FunctionCommon → ( Parameters ) Result Block] (cxt: CONTEXT, env: ENVIRONMENT,
      this: {none, inaccessible}, unchecked: BOOLEAN, prototype: BOOLEAN): UNINSTANTIATEDFUNCTION
   Validate[FunctionCommon](cxt, env, this, unchecked, prototype);
   if prototype then ????
   else
      initialSlots: SLOT{} ← {new SLOT〈〈id: lookupInstanceMember(functionClass,
            QUALIFIEDNAME〈〈namespace: publicNamespace, id: "length"〉〉, read),
            value: realToFloat64(NFixedParameters[Parameters])〉〉};
      return new UNINSTANTIATEDFUNCTION〈〈type: functionClass, defaultSlots: initialSlots, buildPrototype: false,
            call: EvalStaticCall[FunctionCommon], construct: none, instantiations: {}〉〉
   end if
end proc;
```

Setup

```
proc Setup[FunctionDefinition → function FunctionName FunctionCommon] ()
   overriddenIndices: VTABLEINDEX{} ← OverriddenIndices[FunctionDefinition];
   if overriddenIndices = {} then Setup[FunctionCommon]()
   else
      c: CLASS ← EnclosingFrame[FunctionDefinition];
      overriddenSignatures: PARAMETERFRAME{} ← {getInstanceMember(c.super, i).signature |
            ∀i ∈ OverriddenIndices[FunctionDefinition]};
      if |overriddenSignatures| = 1 then
         overriddenSignature: PARAMETERFRAME ← the one element of overriddenSignatures;
         SetupOverride[FunctionCommon](overriddenSignature)
      else throw definitionError
      end if
   end if
end proc;
```

```
proc Setup[FunctionCommon ⇒ ( Parameters ) Result Block] ()
   compileEnv: ENVIRONMENT ← CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   Setup[Parameters](compileEnv, compileFrame);
   Setup[Result](compileEnv, compileFrame);
   Setup[Block]()
end proc;

proc SetupOverride[FunctionCommon ⇒ ( Parameters ) Result Block] (overriddenSignature: PARAMETERFRAME)
   compileEnv: ENVIRONMENT ← CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   SetupOverride[Parameters](compileEnv, compileFrame, overriddenSignature);
   SetupOverride[Result](compileEnv, compileFrame, overriddenSignature);
   Setup[Block]()
end proc;
```

Evaluation

```
proc EvalStaticCall[FunctionCommon ⇒ ( Parameters ) Result Block]
      (this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
   if phase = compile then throw compileExpressionError end if;
   runtimeThis: OBJECTOPT ← none;
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   if compileFrame.prototype then
      g: PACKAGE ∪ GLOBALOBJECT ← getPackageOrGlobalFrame(runtimeEnv);
      if this ∈ {null, undefined} and g ∈ GLOBALOBJECT then runtimeThis ← g
      else runtimeThis ← this
      end if
   end if;
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
   assignArguments(runtimeFrame, args);
   result: OBJECT;
   try Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined); result ← undefined
   catch x: SEMANTICEXCEPTION do
      if x ∈ RETURNEDVALUE then result ← x.value else throw x end if
   end try;
   coercedResult: OBJECT ← runtimeFrame.returnType.implicitCoerce(result, false);
   return coercedResult
end proc;

proc EvalPrototypeConstruct[FunctionCommon ⇒ ( Parameters ) Result Block]
      (args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
   ????
end proc;

proc EvalInstanceCall[FunctionCommon ⇒ ( Parameters ) Result Block]
      (this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
   ????
end proc;
```

```
proc assignArguments(runtimeFrame: ParameterFrame, args: ArgumentList)
  positional: Object[] ← args.positional;
  named: NamedArgument{} ← args.named;
  This procedure performs a number of checks on the arguments, including checking their count, names, and values.
      Although this procedure performs these checks in a specific order for expository purposes, an implementation
      may perform these checks in a different order, which could have the effect of reporting a different error if there
      are multiple errors. For example, if a function only allows between 2 and 4 arguments, the first of which must be
      a Number and is passed five arguments the first of which is a String, then the implementation may throw an
      exception either about the argument count mismatch or about the type coercion error in the first argument.
  if runtimeFrame.unchecked then ???? end if;
  for each parameter ∈ runtimeFrame.positional do
    argument: ObjectOpt;
    if positional = [] then
      argument ← parameter.default;
      if argument = none then throw argumentMismatchError end if
    else argument ← positional[0]; positional ← positional[1 ...]
    end if;
    writeLocalMember(parameter.var, argument, run)
  end for each;
  for each parameter ∈ runtimeFrame.named do
    argument: Object;
    if some na ∈ named satisfies na.name = parameter.name then
      argument ← na.value;
      named ← named – {na}
    else argument ← parameter.default
    end if;
    writeLocalMember(parameter.var, argument, run)
  end for each;
  rest: Variable ∪ {none} ← runtimeFrame.rest;
  if rest = none then
    if positional ≠ [] or named ≠ {} then throw argumentMismatchError end if
  else
    if named ≠ {} and not runtimeFrame.restAllowsNames then
      throw argumentMismatchError
    end if;
    ????
  end if
end proc;
```

Syntax

```
Parameters ⇒
     «empty»
  |  AllParameters

AllParameters ⇒
     Parameter
  |  Parameter , AllParameters
  |  OptionalParameters

OptionalParameters ⇒
     OptionalParameter
  |  OptionalParameter , OptionalParameters
  |  RestAndNamedParameters
```

```
RestAndNamedParameters □
      NamedParameters
   |  RestParameter
   |  RestParameter , NamedParameters
   |  NamedRestParameter

NamedParameters □
      NamedParameter
   |  NamedParameter , NamedParameters

ParameterCore □  TypedIdentifier^allowIn

Parameter □
      ParameterCore
   |  const ParameterCore

OptionalParameter □  Parameter = AssignmentExpression^allowIn

NamedParameterCore □  TypedIdentifier^allowIn = AssignmentExpression^allowIn

NamedParameter □
      named NamedParameterCore
   |  const named NamedParameterCore
   |  named const NamedParameterCore

RestParameter □
      ...
   |  ... Parameter

NamedRestParameter □
      ... named Identifier
   |  ... const named Identifier
   |  ... named const Identifier

Result □
      «empty»
   |  : TypeExpression^allowIn
```

Validation

Plain[*Parameters*]: BOOLEAN;
  Plain[*Parameters* □ «empty»] = **true**;
  Plain[*Parameters* □ *AllParameters*] = Plain[*AllParameters*];

NFixedParameters[*Parameters*]: INTEGER;
  NFixedParameters[*Parameters* □ «empty»] = 0;
  NFixedParameters[*Parameters* □ *AllParameters*] = NFixedParameters[*AllParameters*];

Validate[*Parameters*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Validate to every nonterminal in the expansion of *Parameters*.

Plain[*AllParameters*]: BOOLEAN;
  Plain[*AllParameters* □ *Parameter*] = Plain[*Parameter*];
  Plain[$AllParameters_0$ □ *Parameter* , $AllParameters_1$] = Plain[*Parameter*] **and** Plain[$AllParameters_1$];
  Plain[*AllParameters* □ *OptionalParameters*] = **false**;

```
NFixedParameters[AllParameters]: INTEGER;
   NFixedParameters[AllParameters ⇒ Parameter] = 1;
   NFixedParameters[$AllParameters_0$ ⇒ Parameter , $AllParameters_1$] = 1 + NFixedParameters[$AllParameters_1$];
   NFixedParameters[AllParameters ⇒ OptionalParameters] = NFixedParameters[OptionalParameters];
```

Validate[*AllParameters*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Validate to every nonterminal in the expansion of *AllParameters*.

```
NFixedParameters[OptionalParameters]: INTEGER;
   NFixedParameters[OptionalParameters ⇒ OptionalParameter] = 1;
   NFixedParameters[$OptionalParameters_0$ ⇒ OptionalParameter , $OptionalParameters_1$]
         = 1 + NFixedParameters[$OptionalParameters_1$];
   NFixedParameters[OptionalParameters ⇒ RestAndNamedParameters] = 0;
```

Validate[*OptionalParameters*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Validate to every nonterminal in the expansion of *OptionalParameters*.

Validate[*RestAndNamedParameters*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Validate to every nonterminal in the expansion of *RestAndNamedParameters*.

Validate[*NamedParameters*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Validate to every nonterminal in the expansion of *NamedParameters*.

```
Plain[ParameterCore ⇒ $TypedIdentifier^{allowIn}$]: BOOLEAN = Plain[$TypedIdentifier^{allowIn}$];

CompileVar[ParameterCore]: DYNAMICVAR ∪ VARIABLE;

proc Validate[ParameterCore ⇒ $TypedIdentifier^{allowIn}$]
      (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME, immutable: BOOLEAN)
   Validate[$TypedIdentifier^{allowIn}$](cxt, env);
   name: STRING ← Name[$TypedIdentifier^{allowIn}$];
   v: DYNAMICVAR ∪ VARIABLE;
   if compileFrame.unchecked then
      note not immutable;
      v ← defineHoistedVar(env, name, undefined)
   else
      v ← new VARIABLE⟨⟨type: inaccessible, value: inaccessible, immutable: immutable⟩⟩;
      defineLocalMember(env, name, {publicNamespace}, none, false, readWrite, v)
   end if;
   CompileVar[ParameterCore] ← v
end proc;

Plain[Parameter]: BOOLEAN;
   Plain[Parameter ⇒ ParameterCore] = Plain[ParameterCore];
   Plain[Parameter ⇒ const ParameterCore] = false;

CompileVar[Parameter]: DYNAMICVAR ∪ VARIABLE;

proc Validate[Parameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
   [Parameter ⇒ ParameterCore] do
      Validate[ParameterCore](cxt, env, compileFrame, false);
   [Parameter ⇒ const ParameterCore] do
      Validate[ParameterCore](cxt, env, compileFrame, true)
end proc;
```

```
proc Validate[OptionalParameter ⇒ Parameter = AssignmentExpression^allowIn]
        (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    Validate[Parameter](cxt, env, compileFrame);
    Validate[AssignmentExpression^allowIn](cxt, env)
end proc;

proc Validate[NamedParameterCore ⇒ TypedIdentifier^allowIn = AssignmentExpression^allowIn]
        (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME, immutable: BOOLEAN)
    ????
end proc;

proc Validate[NamedParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [NamedParameter ⇒ named NamedParameterCore] do
        Validate[NamedParameterCore](cxt, env, compileFrame, false);
    [NamedParameter ⇒ const named NamedParameterCore] do
        Validate[NamedParameterCore](cxt, env, compileFrame, true);
    [NamedParameter ⇒ named const NamedParameterCore] do
        Validate[NamedParameterCore](cxt, env, compileFrame, true)
end proc;

proc Validate[RestParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [RestParameter ⇒ ...] do ????;
    [RestParameter ⇒ ... Parameter] do ????
end proc;

proc Validate[NamedRestParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [NamedRestParameter ⇒ ... named Identifier] do ????;
    [NamedRestParameter ⇒ ... const named Identifier] do ????;
    [NamedRestParameter ⇒ ... named const Identifier] do ????
end proc;

Plain[Result]: BOOLEAN;
    Plain[Result ⇒ «empty»] = true;
    Plain[Result ⇒ : TypeExpression^allowIn] = false;
```

Validate[*Result*] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of *Result*.

Setup

Setup[*Parameters*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of *Parameters*.

```
proc SetupOverride[Parameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
        overriddenSignature: PARAMETERFRAME)
    [Parameters ⇒ «empty»] do
        if overriddenSignature.positional ≠ [] or overriddenSignature.named ≠ {} or
                overriddenSignature.rest ≠ none then
            throw definitionError
        end if;
    [Parameters ⇒ AllParameters] do
        SetupOverride[AllParameters](compileEnv, compileFrame, overriddenSignature, overriddenSignature.positional)
end proc;
```

```
proc Setup[AllParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
  [AllParameters □ Parameter] do Setup[Parameter](compileEnv, compileFrame, none);
  [AllParameters$_0$ □ Parameter , AllParameters$_1$] do
    Setup[Parameter](compileEnv, compileFrame, none);
    Setup[AllParameters$_1$](compileEnv, compileFrame);
  [AllParameters □ OptionalParameters] do
    Setup[OptionalParameters](compileEnv, compileFrame)
end proc;

proc SetupOverride[AllParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
    overriddenSignature: PARAMETERFRAME, overriddenPositional: PARAMETER[])
  [AllParameters □ Parameter] do
    if overriddenPositional = [] then throw definitionError end if;
    SetupOverride[Parameter](compileEnv, compileFrame, none, overriddenPositional[0]);
    if |overriddenPositional| $\neq$ 1 or overriddenSignature.named $\neq$ {} or overriddenSignature.rest $\neq$ none then
      throw definitionError
    end if;
  [AllParameters$_0$ □ Parameter , AllParameters$_1$] do
    if overriddenPositional = [] then throw definitionError end if;
    SetupOverride[Parameter](compileEnv, compileFrame, none, overriddenPositional[0]);
    SetupOverride[AllParameters$_1$](compileEnv, compileFrame, overriddenSignature, overriddenPositional[1 ...]);
  [AllParameters □ OptionalParameters] do
    SetupOverride[OptionalParameters](compileEnv, compileFrame, overriddenSignature, overriddenPositional)
end proc;
```

Setup[*OptionalParameters*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of *OptionalParameters*.

```
proc SetupOverride[OptionalParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
    overriddenSignature: PARAMETERFRAME, overriddenPositional: PARAMETER[])
  [OptionalParameters □ OptionalParameter] do
    if overriddenPositional = [] then throw definitionError end if;
    SetupOverride[OptionalParameter](compileEnv, compileFrame, overriddenPositional[0]);
    if |overriddenPositional| $\neq$ 1 or overriddenSignature.named $\neq$ {} or overriddenSignature.rest $\neq$ none then
      throw definitionError
    end if;
  [OptionalParameters$_0$ □ OptionalParameter , OptionalParameters$_1$] do
    if overriddenPositional = [] then throw definitionError end if;
    SetupOverride[OptionalParameter](compileEnv, compileFrame, overriddenPositional[0]);
    SetupOverride[OptionalParameters$_1$](compileEnv, compileFrame, overriddenSignature,
        overriddenPositional[1 ...]);
  [OptionalParameters □ RestAndNamedParameters] do
    if overriddenPositional $\neq$ [] then throw definitionError end if;
    SetupOverride[RestAndNamedParameters](compileEnv, compileFrame, overriddenSignature)
end proc;
```

Setup[*RestAndNamedParameters*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of *RestAndNamedParameters*.

```
proc SetupOverride[RestAndNamedParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
    overriddenSignature: PARAMETERFRAME)
  [RestAndNamedParameters □ NamedParameters] do
    if overriddenSignature.rest $\neq$ none then throw definitionError end if;
    SetupOverride[NamedParameters](compileEnv, compileFrame, overriddenSignature.named);
    [RestAndNamedParameters □ RestParameter] do
        rest: VARIABLE □ {none} □ overriddenSignature.rest;
        if rest = none or overriddenSignature.restAllowsNames or overriddenSignature.named ≠ {} then
            throw definitionError
        end if;
        SetupOverride[RestParameter](compileEnv, compileFrame, rest);
    [RestAndNamedParameters □ RestParameter , NamedParameters] do
        rest: VARIABLE □ {none} □ overriddenSignature.rest;
        if rest = none or overriddenSignature.restAllowsNames then throw definitionError
        end if;
        SetupOverride[RestParameter](compileEnv, compileFrame, rest);
        SetupOverride[NamedParameters](compileEnv, compileFrame, overriddenSignature.named);
    [RestAndNamedParameters □ NamedRestParameter] do
        rest: VARIABLE □ {none} □ overriddenSignature.rest;
        if rest = none or not overriddenSignature.restAllowsNames or overriddenSignature.named ≠ {} then
            throw definitionError
        end if;
        SetupOverride[NamedRestParameter](compileEnv, compileFrame, rest)
end proc;
```

Setup[*NamedParameters*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of *NamedParameters*.

```
proc SetupOverride[NamedParameters]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenNamed: NAMEDPARAMETER{})
    [NamedParameters □ NamedParameter] do
        remainingNamed: NAMEDPARAMETER{} □
                SetupOverride[NamedParameter](compileEnv, compileFrame, overriddenNamed);
        if remainingNamed ≠ {} then throw definitionError end if;
    [NamedParameters_0 □ NamedParameter , NamedParameters_1] do
        remainingNamed: NAMEDPARAMETER{} □
                SetupOverride[NamedParameter](compileEnv, compileFrame, overriddenNamed);
        SetupOverride[NamedParameters_1](compileEnv, compileFrame, remainingNamed)
end proc;

proc Setup[ParameterCore □ TypedIdentifier^allowIn]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, default: OBJECTOPT)
    v: DYNAMICVAR □ VARIABLE □ CompileVar[ParameterCore];
    case v of
        DYNAMICVAR do nothing;
        VARIABLE do
            type: CLASSOPT □ SetupAndEval[TypedIdentifier^allowIn](compileEnv);
            if type = none then type □ objectClass end if;
            v.type □ type;
            v.value □ uninitialised
    end case;
    p: PARAMETER □ PARAMETER□var: v, default: default□
    compileFrame.positional □ compileFrame.positional ⊕ [p]
end proc;
```

```
proc SetupOverride[ParameterCore □ TypedIdentifier$^{\text{allowIn}}$] (compileEnv: ENVIRONMENT,
        compileFrame: PARAMETERFRAME, default: OBJECTOPT, overriddenParameter: PARAMETER)
    v: DYNAMICVAR □ VARIABLE □ CompileVar[ParameterCore];
    note v □ DYNAMICVAR;
    type: CLASSOPT □ SetupAndEval[TypedIdentifier$^{\text{allowIn}}$](compileEnv);
    if type = none then type □ objectClass end if;
    if type $\neq$ getVariableType(overriddenParameter.var, compile) then
        throw definitionError
    end if;
    v.type □ type;
    v.value □ uninitialised;
    newDefault: OBJECTOPT □ default;
    if newDefault = none then newDefault □ overriddenParameter.default end if;
    p: PARAMETER □ PARAMETER□var: v, default: newDefault□
    compileFrame.positional □ compileFrame.positional $\oplus$ [p]
end proc;
```

Setup[*Parameter*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME, *default*: OBJECTOPT) propagates the call to Setup to every nonterminal in the expansion of *Parameter*.

SetupOverride[*Parameter*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME, *default*: OBJECTOPT, *overriddenParameter*: PARAMETER) propagates the call to SetupOverride to every nonterminal in the expansion of *Parameter*.

```
proc Setup[OptionalParameter □ Parameter = AssignmentExpression$^{\text{allowIn}}$]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    Setup[AssignmentExpression$^{\text{allowIn}}$]();
    default: OBJECT □ readReference(Eval[AssignmentExpression$^{\text{allowIn}}$](compileEnv, compile), compile);
    Setup[Parameter](compileEnv, compileFrame, default)
end proc;

proc SetupOverride[OptionalParameter □ Parameter = AssignmentExpression$^{\text{allowIn}}$]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenParameter: PARAMETER)
    Setup[AssignmentExpression$^{\text{allowIn}}$]();
    default: OBJECT □ readReference(Eval[AssignmentExpression$^{\text{allowIn}}$](compileEnv, compile), compile);
    SetupOverride[Parameter](compileEnv, compileFrame, default, overriddenParameter)
end proc;

proc Setup[NamedParameterCore □ TypedIdentifier$^{\text{allowIn}}$ = AssignmentExpression$^{\text{allowIn}}$]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    ????
end proc;

proc SetupOverride[NamedParameterCore □ TypedIdentifier$^{\text{allowIn}}$ = AssignmentExpression$^{\text{allowIn}}$]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenNamed: NAMEDPARAMETER{}):
        NAMEDPARAMETER{}
    ????
end proc;
```

Setup[*NamedParameter*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of *NamedParameter*.

```
proc SetupOverride[NamedParameter]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenNamed: NAMEDPARAMETER{}):
        NAMEDPARAMETER{}
    [NamedParameter □ named NamedParameterCore] do
        return SetupOverride[NamedParameterCore](compileEnv, compileFrame, overriddenNamed);
    [NamedParameter  const named NamedParameterCore] do
        return SetupOverride[NamedParameterCore](compileEnv, compileFrame, overriddenNamed);
    [NamedParameter  named const NamedParameterCore] do
        return SetupOverride[NamedParameterCore](compileEnv, compileFrame, overriddenNamed)
end proc;

proc Setup[RestParameter] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [RestParameter  ...] do ????;
    [RestParameter  ... Parameter] do ????
end proc;

proc SetupOverride[RestParameter]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenRest: VARIABLE)
    [RestParameter  ...] do ????;
    [RestParameter  ... Parameter] do ????
end proc;

proc Setup[NamedRestParameter] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [NamedRestParameter  ... named Identifier] do ????;
    [NamedRestParameter  ... const named Identifier] do ????;
    [NamedRestParameter  ... named const Identifier] do ????
end proc;

proc SetupOverride[NamedRestParameter]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenRest: VARIABLE)
    [NamedRestParameter  ... named Identifier] do ????;
    [NamedRestParameter  ... const named Identifier] do ????;
    [NamedRestParameter  ... named const Identifier] do ????
end proc;

proc Setup[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [Result  «empty»] do compileFrame.returnType  objectClass;
    [Result  : TypeExpression$^{allowIn}$] do
        compileFrame.returnType  SetupAndEval[TypeExpression$^{allowIn}$](compileEnv)
end proc;

proc SetupOverride[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
        overriddenSignature: PARAMETERFRAME)
    [Result  «empty»] do compileFrame.returnType  overriddenSignature.returnType;
    [Result  : TypeExpression$^{allowIn}$] do
        t: CLASS  SetupAndEval[TypeExpression$^{allowIn}$](compileEnv);
        if overriddenSignature.returnType ≠ t then throw definitionError end if;
        compileFrame.returnType  t
end proc;
```

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⟨⟨localBindings: {}, parent: superclass, instanceBindings: {}, vTable: {},
         instanceInitOrder: [], complete: false, super: superclass, prototype: prototype, typeofString: "object",
         privateNamespace: privateNamespace, dynamic: dynamic, final: final, call: call, construct: construct,
         defaultValue: null⟩⟩;
   proc isInstanceOf(o: OBJECT): BOOLEAN
      return o = null or isAncestor(c, objectType(o))
   end proc;
   proc coerce(o: OBJECT, silent: BOOLEAN): OBJECT
      if c.isInstanceOf(o) then return o
      elsif silent then return null
      else throw badValueError
      end if
   end proc;
   c.isInstanceOf ← isInstanceOf;
   c.implicitCoerce ← coerce;
   Class[ClassDefinition] ← c;
   v: VARIABLE ← new VARIABLE⟨⟨type: classClass, value: c, immutable: true⟩⟩;
   defineLocalMember(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 SetupAndEval[TypeExpression^allowIn](env)
end proc;
```

Setup

```
proc Setup[ClassDefinition ⇒ class Identifier Inheritance Block] ()
   Setup[Block]()
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⟩⟩;
   defineLocalMember(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
      Validate[Directives](initialContext, initialEnvironment, JUMPTARGETS⟨⟨breakTargets: {}, continueTargets: {}⟩⟩,
            singular, none);
      Setup[Directives]();
      return Eval[Directives](initialEnvironment, undefined)
   end;
```

17 Predefined Identifiers

18 Built-in Classes

```
proc makeBuiltInClass(superclass: CLASSOPT, typeofString: STRING, dynamic: BOOLEAN, allowNull: BOOLEAN,
      final: BOOLEAN, defaultValue: OBJECT): CLASS
   proc call(this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
      ????
   end proc;
   proc construct(args: ARGUMENTLIST, phase: PHASE): OBJECT
      ????
   end proc;
   proc implicitCoerce(o: OBJECT, silent: BOOLEAN): OBJECT
      ????
   end proc;
   privateNamespace: NAMESPACE ← new NAMESPACE⟨⟨name: "private"⟩⟩;
   c: CLASS ← new CLASS⟨⟨localBindings: {}, parent: superclass, instanceBindings: {}, vTable: {},
         instanceInitOrder: [], complete: true, super: superclass, prototype: null, typeofString: typeofString,
         privateNamespace: privateNamespace, dynamic: dynamic, final: final, call: call, construct: construct,
         implicitCoerce: implicitCoerce, defaultValue: defaultValue⟩⟩;
   proc isInstanceOf(o: OBJECT): BOOLEAN
      return isAncestor(c, objectType(o)) or (allowNull and o = null)
   end proc;
   c.isInstanceOf ← isInstanceOf;
   return c
end proc;
```

```
proc makeBuiltInIntegerClass(low: INTEGER, high: INTEGER): CLASS
   proc call(this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
      ????
   end proc;
   proc construct(args: ARGUMENTLIST, phase: PHASE): OBJECT
      ????
   end proc;
   proc isInstanceOf(o: OBJECT): BOOLEAN
      if o ∈ FLOAT64 then
         case o of
            {$\text{NaN64}_{f64}$, $+\infty_{f64}$, $-\infty_{f64}$} do return false;
            {$+\text{zero}_{f64}$, $-\text{zero}_{f64}$} do return true;
            NONZEROFINITEFLOAT64 do
               r: RATIONAL ← o.value;
               if r ∈ INTEGER and $low \le r \le high$ then return true
               else return false
               end if
         end case
      else return false
      end if
   end proc;
   proc implicitCoerce(o: OBJECT, silent: BOOLEAN): OBJECT
      if o = undefined then return $+\text{zero}_{f64}$
      elsif o ∈ GENERALNUMBER then
         i: INTEGEROPT ← checkInteger(o);
         if $i \ne$ none and $low \le i \le high$ then
            note $-\text{zero}_{f32}$, $+\text{zero}_{f32}$, and $-\text{zero}_{f64}$ are all coerced to $+\text{zero}_{f64}$.
            return realToFloat64(i)
         end if
      end if;
      throw badValueError
   end proc;
   privateNamespace: NAMESPACE ← new NAMESPACE⟨⟨name: "private"⟩⟩;
   return new CLASS⟨⟨localBindings: {}, parent: numberClass, instanceBindings: {}, vTable: {},
         instanceInitOrder: [], complete: true, super: numberClass, prototype: null, typeofString: "number",
         privateNamespace: privateNamespace, dynamic: false, final: true, call: call, construct: construct,
         isInstanceOf: isInstanceOf, implicitCoerce: implicitCoerce, defaultValue: $+\text{zero}_{f64}$⟩⟩
end proc;
```

objectClass: CLASS = makeBuiltInClass(**none**, "`object`", **false**, **true**, **false**, **undefined**);

undefinedClass: CLASS = makeBuiltInClass(objectClass, "`undefined`", **false**, **false**, **true**, **undefined**);

nullClass: CLASS = makeBuiltInClass(objectClass, "`object`", **false**, **true**, **true**, **null**);

booleanClass: CLASS = makeBuiltInClass(objectClass, "`boolean`", **false**, **false**, **true**, **false**);

generalNumberClass: CLASS = makeBuiltInClass(objectClass, "`object`", **false**, **false**, **false**, NaN64_{f64});

longClass: CLASS = makeBuiltInClass(generalNumberClass, "`long`", **false**, **false**, **true**, LONG⟨⟨value: 0⟩⟩);

uLongClass: CLASS = makeBuiltInClass(generalNumberClass, "`ulong`", **false**, **false**, **true**, ULONG⟨⟨value: 0⟩⟩);

floatClass: CLASS = makeBuiltInClass(generalNumberClass, "`float`", **false**, **false**, **true**, NaN32_{f32});

numberClass: CLASS = makeBuiltInClass(generalNumberClass, "`number`", **false**, **false**, **true**, NaN64_{f64});

sByteClass: CLASS = makeBuiltInIntegerClass(–128, 127);

byteClass: CLASS = *makeBuiltInIntegerClass*(0, 255);

shortClass: CLASS = *makeBuiltInIntegerClass*(–32768, 32767);

uShortClass: CLASS = *makeBuiltInIntegerClass*(0, 65535);

intClass: CLASS = *makeBuiltInIntegerClass*(–2147483648, 2147483647);

uIntClass: CLASS = *makeBuiltInIntegerClass*(0, 4294967295);

characterClass: CLASS = *makeBuiltInClass*(*objectClass*, “`character`”, **false**, **false**, **true**, ‘«NUL»’);

stringClass: CLASS = *makeBuiltInClass*(*objectClass*, “`string`”, **false**, **true**, **true**, **null**);

namespaceClass: CLASS = *makeBuiltInClass*(*objectClass*, “`namespace`”, **false**, **true**, **true**, **null**);

attributeClass: CLASS = *makeBuiltInClass*(*objectClass*, “`object`”, **false**, **true**, **true**, **null**);

dateClass: CLASS = *makeBuiltInClass*(*objectClass*, “`object`”, **true**, **true**, **true**, **null**);

regExpClass: CLASS = *makeBuiltInClass*(*objectClass*, “`object`”, **true**, **true**, **true**, **null**);

classClass: CLASS = *makeBuiltInClass*(*objectClass*, “`function`”, **false**, **true**, **true**, **null**);

functionClass: CLASS = *makeBuiltInClass*(*objectClass*, “`function`”, **false**, **true**, **true**, **null**);

prototypeClass: CLASS = *makeBuiltInClass*(*objectClass*, “`object`”, **true**, **true**, **true**, **null**);

packageClass: CLASS = *makeBuiltInClass*(*objectClass*, “`object`”, **true**, **true**, **true**, **null**);

objectPrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE⟨⟨localBindings: {}, parent: **none**, sealed: **false**,
    type: *prototypeClass*, slots: {}, call: **none**, construct: **none**, toClass: **none**, env: **none**⟩⟩;

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 Error

18.17 Attribute

19 Built-in Functions

20 Built-in Attributes

21 Built-in Namespaces

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

A Index

A.1 Nonterminals

A.2 Tags

A.3 Semantic Domains

A.4 Globals