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* ≠ *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** $x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}$

$\{x^2 \mid \forall x \in \{-5, -1, 1, 2, 4\}\} = \{1, 4, 16, 25\}$

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

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
$\|A\|$	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 $x \in A$, $x \geq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
max *A*	The value *m* that satisfies both $m \in A$ and for all elements $x \in A$, $x \leq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)
$A \cap B$	The intersection of *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*.
$A \neq B$	**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** $A \neq B$.

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

T = {1,2,3}

then:

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

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

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

some $x \in A$ **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 $\subset$ 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 $\subset$ REAL. π is an example of a real number slightly larger than 3.14.

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

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

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

5.6.1 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:

..., 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 $\lfloor x \cdot 2^{count} \rfloor$

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 **|** $\forall x \in$ **[**–1, 1, 2, 3, 4, 2, 5**]]** = **[**1, 1, 4, 9, 16, 4, 25**]**
[x+1 **|** $\forall x \in$ **[**–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, ... , e_{n-1}**]** and v = **[**f_0, f_1, ... , 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
$\lvert u \rvert$		The length n of the list
$u[i]$	$0 \le i < \lvert u \rvert$	The i^{th} element e_i.
$u[i \ldots j]$	$0 \le i \le j+1 \le \lvert u \rvert$	The list slice [e_i, e_{i+1}, ... , e_j] consisting of all elements of u between the i^{th} and the j^{th}, inclusive. The result is the empty list [] if j=i–1.
$u[i \ldots]$	$0 \le i \le \lvert u \rvert$	The list slice [e_i, e_{i+1}, ... , e_{n-1}] consisting of all elements of u between the i^{th} and the end. The result is the empty list [] if i=n.
$u[i \setminus x]$	$0 \le i < \lvert u \rvert$	The list [e_0, ... , e_{i-1}, x, e_{i+1}, ... , e_{n-1}] with the i^{th} element replaced by the value x and the other elements unchanged
$u \oplus v$		The concatenated list [e_0, e_1, ... , e_{n-1}, f_0, f_1, ... , f_{m-1}]
repeat(e, i)	i ≥ 0	The list **[**e, e, ... , 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 i^{th} field's value v_i.

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

5.10.1 Shorthand Notation

The semantic notation *ns*::*id* is a shorthand for QUALIFIEDNAME⟨namespace: *ns*, id: *id*⟩. See section 9.1.6.1.

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

5.12.1.1 Shorthand Notation

In this specification, when i is an integer between -2^{63} and $2^{63} - 1$, the notation $i_{\mathbf{long}}$ indicates the result of LONG□value: i□, which is the integer i wrapped in a LONG tuple.

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

5.12.2.1 Shorthand Notation

In this specification, when *i* is an integer between 0 and $2^{64} - 1$, the notation i_{ulong} indicates the result of ULONG⟦value: *i*⟧ which is the integer *i* wrapped in a ULONG tuple.

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 ∪ $\{+\infty_{f32}, -\infty_{f32}, \text{NaN}_{f32}\}$;
FINITEFLOAT32 = NONZEROFINITEFLOAT32 ∪ $\{+\text{zero}_{f32}, -\text{zero}_{f32}\}$

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

m is called the significand.

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

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

m is called the significand.

The remaining FLOAT32 values are the tags $+\text{zero}_{f32}$ (positive zero), $-\text{zero}_{f32}$ (negative zero), $+\infty_{f32}$ (positive infinity), $-\infty_{f32}$ (negative infinity), and NaN_{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 $+\text{zero}_{f32} \neq -\text{zero}_{f32}$ but $\text{NaN}_{f32} = \text{NaN}_{f32}$. 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_{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_{f32} is a FLOAT32 value (whose exact value is actually 3.400000095367431640625). The positive finite FLOAT32 values range from $10^{-45}{}_{f32}$ to $(3.4028235 \times 10^{38})_{f32}$.

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 $a \neq 0$ **then return** NONZEROFINITEFLOAT32⟨value: *a*⟩
  elsif $x < 0$ **then return** $-\mathbf{zero}_{f32}$
  else return $+\mathbf{zero}_{f32}$
  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 $x \in \{+\mathbf{zero}_{f32}, -\mathbf{zero}_{f32}\}$ **then return** 0 **end if**;
  r: RATIONAL ← *x*.value;
  if $r > 0$ **then return** $\lfloor r \rfloor$ **else return** $\lceil r \rceil$ **end if**
end proc

5.12.3.3 Arithmetic

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

float32Negate(*x*: FLOAT32): FLOAT32

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

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 ∪ $\{+\infty_{f64}, -\infty_{f64}, \mathbf{NaN}_{f64}\}$;
FINITEFLOAT64 = NONZEROFINITEFLOAT64 ∪ $\{+\mathbf{zero}_{f64}, -\mathbf{zero}_{f64}\}$

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

m is called the significand.

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

DENORMALISEDFLOAT64VALUES = $\{s \cdot m \cdot 2^{-1074} \mid \forall s \in \{-1, 1\}, \forall m \in \{1 \ldots 2^{52}-1\}\}$

m is called the significand.

The remaining FLOAT64 values are the tags **+zero**$_{\textbf{f64}}$ (positive zero), **–zero**$_{\textbf{f64}}$ (negative zero), **+∞**$_{\textbf{f64}}$ (positive infinity), **–∞**$_{\textbf{f64}}$ (negative infinity), and **NaN**$_{\textbf{f64}}$ (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 **+zero**$_{\textbf{f64}}$ ≠ **–zero**$_{\textbf{f64}}$ but **NaN**$_{\textbf{f64}}$ = **NaN**$_{\textbf{f64}}$. 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 $x_{\textbf{f64}}$ indicates the result of *realToFloat64*(*x*), which is the "closest" FLOAT64 value as defined below. Thus, 3.4 is a REAL number, while $3.4_{\textbf{f64}}$ is a FLOAT64 value (whose exact value is actually 3.399999999999999911182158029987476766109466552734375). The positive finite FLOAT64 values range from $(5 \times 10^{-324})_{\textbf{f64}}$ to $(1.7976931348623157 \times 10^{308})_{\textbf{f64}}$.

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 +∞**$_{\textbf{f64}}$
  elsif *a* = -2^{1024} **then return –∞**$_{\textbf{f64}}$
  elsif *a* ≠ 0 **then return** NONZEROFINITEFLOAT64⟦value: *a*⟧
  elsif *x* < 0 **then return –zero**$_{\textbf{f64}}$
  else return +zero$_{\textbf{f64}}$
  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
–∞$_{\textbf{f32}}$	**–∞**$_{\textbf{f64}}$
–zero$_{\textbf{f32}}$	**–zero**$_{\textbf{f64}}$
+zero$_{\textbf{f32}}$	**+zero**$_{\textbf{f64}}$
+∞$_{\textbf{f32}}$	**+∞**$_{\textbf{f64}}$
NaN$_{\textbf{f32}}$	**NaN**$_{\textbf{f64}}$
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* ∈ {**+zero**$_{\textbf{f64}}$, **–zero**$_{\textbf{f64}}$} **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 $(\mathit{expr})_{\text{f64}}$ is a shorthand for *realToFloat64*(*expr*).

float64Abs(*x*: FLOAT64): FLOAT64

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

float64Negate(*x*: FLOAT64): FLOAT64

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

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

x \ *y*	$-\infty_{\text{f64}}$	negative finite	$-\text{zero}_{\text{f64}}$	$+\text{zero}_{\text{f64}}$	positive finite	$+\infty_{\text{f64}}$	NaN_{f64}
$-\infty_{\text{f64}}$	$-\infty_{\text{f64}}$	$-\infty_{\text{f64}}$	$-\infty_{\text{f64}}$	$-\infty_{\text{f64}}$	$-\infty_{\text{f64}}$	NaN_{f64}	NaN_{f64}
negative finite	$-\infty_{\text{f64}}$	$(x.\text{value} + y.\text{value})_{\text{f64}}$	*x*	*x*	$(x.\text{value} + y.\text{value})_{\text{f64}}$	$+\infty_{\text{f64}}$	NaN_{f64}
$-\text{zero}_{\text{f64}}$	$-\infty_{\text{f64}}$	*y*	$-\text{zero}_{\text{f64}}$	$+\text{zero}_{\text{f64}}$	*y*	$+\infty_{\text{f64}}$	NaN_{f64}
$+\text{zero}_{\text{f64}}$	$-\infty_{\text{f64}}$	*y*	$+\text{zero}_{\text{f64}}$	$+\text{zero}_{\text{f64}}$	*y*	$+\infty_{\text{f64}}$	NaN_{f64}
positive finite	$-\infty_{\text{f64}}$	$(x.\text{value} + y.\text{value})_{\text{f64}}$	*x*	*x*	$(x.\text{value} + y.\text{value})_{\text{f64}}$	$+\infty_{\text{f64}}$	NaN_{f64}
$+\infty_{\text{f64}}$	NaN_{f64}	$+\infty_{\text{f64}}$	$+\infty_{\text{f64}}$	$+\infty_{\text{f64}}$	$+\infty_{\text{f64}}$	$+\infty_{\text{f64}}$	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 $-\text{zero}_{\text{f64}}$, not $+\text{zero}_{\text{f64}}$.

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

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

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

	y			
x	$-\infty_{f64}$, $+\infty_{f64}$	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*	$(x.\text{value} - \lvert y.\text{value}\rvert \cdot \lfloor x.\text{value}/\lvert y.\text{value}\rvert \rfloor)_{f64}$	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*(*param*$_1$: T$_1$, … , *param*$_n$: T$_n$): T
  step$_1$;
  step$_2$;
  … ;
  step$_m$
end proc;

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

f is the procedure's name, *param*$_1$ through *param*$_n$ are the procedure's parameters, T_1 through T_n are the parameters' respective semantic domains, T is the semantic domain of the procedure's result, and *step*$_1$ through *step*$_m$ describe the procedure's computation steps, which may produce side effects and/or return a result. If T is omitted, the procedure does not return a result. When the procedure is called with argument values v_1 through v_n, the procedure's steps are performed and the result, if any, returned to the caller.

A procedure's steps can refer to the parameters *param*$_1$ through *param*$_n$; each reference to a parameter *param*$_i$ evaluates to the corresponding argument value v_i. Procedure parameters are statically scoped. Arguments are passed by value.

5.13.1 Operations

The only operation done on a procedure *f* is calling it using the *f*(*arg*$_1$, ..., *arg*$_n$) syntax. *f* is computed first, followed by the argument expressions *arg*$_1$ through *arg*$_n$, in left-to-right order. If the result of computing *f* or any of the argument expressions throws an exception *e*, then the call immediately propagates *e* without computing any following argument expressions. Otherwise, *f* is invoked using the provided arguments and the resulting value, if any, returned to the caller.

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

5.13.2 Semantic Domains of Procedures

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

5.13.3 Steps

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

nothing

A **nothing** step performs no operation.

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**
  T_1 **do** *step*; *step*; ...; *step*;
  T_2 **do** *step*; *step*; ...; *step*;
  ...;
  T_n **do** *step*; *step*; ...; *step*
  else *step*; *step*; ...; *step*
end case

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

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

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

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

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

return *expression*

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

invariant *expression*

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

throw *expression*

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

```
try
   step;
   step;
   …;
   step
catch v: T do
   step;
   step;
   …;
   step
end try
```

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

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

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

expands into the following four rules:

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

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

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

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

AssignmentExpression$^{\mathrm{normal,allowIn}}$ 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* ≥ 2 **and** *base* ≤ 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 ⇒ Digits₁ # Digits₂]
   begin
      base: INTEGER ← DecimalValue[Digits₂];
      if base ≥ 2 and base ≤ 10 then return BaseValue[Digits₁](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;
   [Digits₀ ⇒ Digits₁ Digit] do
      d: INTEGER ← Value[Digit];
      if d < base then return base×BaseValue[Digits₁](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* □ *expression*]
  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`, `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 $NextInputElement^{re}$, $NextInputElement^{div}$, and $NextInputElement^{num}$, 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 $NextInputElement^{num}$ if the previous lexed token was a **Number** or **NegatedMinLong**, $NextInputElement^{re}$ if the previous token was not a **Number** or **NegatedMinLong** and a `/` should be interpreted as starting a regular expression, and $NextInputElement^{div}$ 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

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

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

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

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

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

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

Semantics

The grammar parameter □ can be either re or div.

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

Lex[*InputElement*$^{□}$]: INPUTELEMENT;
  Lex[*InputElement*$^{□}$ □ *LineBreaks*] = **LineBreak**;
  Lex[*InputElement*$^{□}$ □ *IdentifierOrKeyword*] = Lex[*IdentifierOrKeyword*];
  Lex[*InputElement*$^{□}$ □ *Punctuator*] = Lex[*Punctuator*];
  Lex[*InputElement*$^{\text{div}}$ □ *DivisionPunctuator*] = Lex[*DivisionPunctuator*];
  Lex[*InputElement*$^{□}$ □ *NumericLiteral*] = Lex[*NumericLiteral*];
  Lex[*InputElement*$^{□}$ □ *StringLiteral*] = Lex[*StringLiteral*];
  Lex[*InputElement*$^{\text{re}}$ □ *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`", "`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`, 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*]□$10^{\text{LexNumber}[\textit{SignedInteger}]}$;

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*]/$10^{\text{NDigits}[\textit{DecimalDigits}]}$;

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, or a package 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;

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**};

Some 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**. Semantics that access variables that could be in the **uninitialised** state check whether the value read is, in fact, **uninitialised** and generally throw an exception in that case.

ObjectU = Object ∪ {**uninitialised**};

The semantic domain ObjectUOpt consists of all objects as well as the tags **none** and **uninitialised**:

ObjectUOpt = Object ∪ {**uninitialised**, **none**};

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.

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

The semantic notation *ns*::*id* is a shorthand for QUALIFIEDNAME⟦namespace: *ns*, id: *id*⟧.

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
enumerable	BOOLEAN	**true** if the `enumerable` 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 *****)
super	CLASSOPT	This class's immediate superclass or **null** if none
instanceMembers	INSTANCEMEMBER{}	Map of qualified names to instance members defined or overridden in this class
complete	BOOLEAN	**true** after all members of this class have been added to this CLASS record
prototype	OBJECTOPT	The default value of the `super` field of newly created simple instances of this class; **none** for most classes
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
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.
bracketRead	OBJECT × CLASS × OBJECT[] × PHASE → OBJECTOPT	
bracketWrite	OBJECT × CLASS × OBJECT[] × OBJECT × {**run**} → {**none**, **ok**}	
bracketDelete	OBJECT × CLASS × OBJECT[] × {**run**} → BOOLEANOPT	
read	OBJECT × CLASS × MULTINAME × LOOKUPKIND × PHASE → OBJECTOPT	
write	OBJECT × CLASS × MULTINAME × LOOKUPKIND × BOOLEAN × OBJECT × {**run**} → {**none**, **ok**}	
delete	OBJECT × CLASS × MULTINAME × LOOKUPKIND × {**run**} → BOOLEANOPT	
enumerate	OBJECT → OBJECT{}	
call	OBJECT × OBJECT[] × PHASE → OBJECT	A procedure to call when this class is used in a call expression. The parameters are the `this` argument, the list of arguments, and the phase of evaluation (section 9.5).
construct	OBJECT[] × PHASE → OBJECT	A procedure to call when this class is used in a `new` expression. The parameters are the list of arguments and the phase of evaluation (section 9.5).

is	OBJECT □ BOOLEAN	A procedure to call to determine whether a given object i instance of this class
implicitCoerce	OBJECT □ BOOLEAN □ OBJECT	A procedure to call when a value is assigned to a varia parameter, or result whose type is this class. The argumen implicitCoerce can be any value, which may or may not b instance of this class; the result must be an instance of this clas the coercion is not appropriate, implicitCoerce should throw exception if its second argument is **false** or return **null** (as lon **null** is an instance of this class) if its second argument is **true**.

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

CLASSOPT = CLASS □ {**none**}

CLASSU consists of all classes as well as **uninitialised**:

CLASSU = CLASS □ {**uninitialised**};

A CLASS *c* is an *ancestor* of CLASS *d* if either *c* = *d* or *d*.super = *s*, *s* ≠ **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* ≠ *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
super	OBJECTOPT	Optional link to the next object in this instance's prototype chain
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 □ SIMPLEINSTANCE □ OBJECT[] □ 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), a SIMPLEINSTANCE (the called instance), a list of OBJECT argument values, and a PHASE (see section 9.5) and produces an OBJECT result
construct	SIMPLEINSTANCE □ OBJECT[] □ PHASE □ OBJECT □ {**none**}	Either **none** or a procedure to call when this instance is used in a `new` expression. The procedure takes a SIMPLEINSTANCE (the instance on which `new` was invoked), a list of OBJECT argument values, and a PHASE (see section 9.5) and produces an OBJECT result
env	ENVIRONMENTOPT	Either **none** or the environment in which call or construct should look up non-local variables

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

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
buildPrototype	BOOLEAN	If true, the generated SIMPLEINSTANCE gets a separate `prototype` property with its own protype object
length	INTEGER	The value to store in the generated SIMPLEINSTANCE's `length` property
call	OBJECT □ SIMPLEINSTANCE □ OBJECT[] □ PHASE □ OBJECT □ {**none**}	Values to be transferred into the generated SIMPLEINSTANCE's corresponding fields
construct	SIMPLEINSTANCE □ OBJECT[] □ 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)
super	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)
super	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 and Global Objects

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

Field	Contents	Note
localBindings	LOCALBINDING{}	Same as in SIMPLEINSTANCEs (section 9.1.9)
super	OBJECTOPT	
sealed	BOOLEAN	
internalNamespace	NAMESPACE	This package's or 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 one of the limit class's descendants.
limit	CLASS	The immediate superclass of 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	OBJECT	The object whose property was referenced (*a* in the examples above).
limit	CLASS	The most specific class to consider when searching for properties of the object *a*. Normally limit is *a*'s class, but can be one of that class's ancestors if *a* is a `super` expression.
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	OBJECT	The object whose property was referenced (*a* in the examples above).
limit	CLASS	The most specific class to consider when searching for properties of the object *a*. Normally limit is *a*'s class, but can be one of that class's ancestors if *a* is a `super` expression.
args	OBJECT[]	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 Phases of 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 phases of expression evaluation:

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

9.6 Contexts

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

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

		always a member of this set.
constructsSuper	BOOLEANOPT	A flag that indicates whether a call to another constructor has been detected yet during static analysis of a class constructor. constructsSuper is **none** outside class constructors.

9.7 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

9.8 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. A WITHFRAME is always preceded by a LOCALFRAME, so the first frame is never a WITHFRAME.

ENVIRONMENT = FRAME[]

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

ENVIRONMENTU = ENVIRONMENT ∪ {**uninitialised**};

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 package, a function parameter frame, a class, a local (block) frame, or a `with` statement frame:

FRAME = NONWITHFRAME ∪ WITHFRAME;

NONWITHFRAME = SYSTEMFRAME ∪ PACKAGE ∪ PARAMETERFRAME ∪ CLASS ∪ LOCALFRAME;

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**}

9.8.1 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

9.8.2 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 9.8
this	ObjectUOpt	The value of `this`; **none** if this function doesn't define `this`; **uninitialised** 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
parameters	Parameter[]	List of this function's parameters
rest	VariableOpt	The parameter variable for collecting any extra arguments that may be passed or **none** if no extra arguments are allowed
returnType	Class	The function's declared return type, which defaults to `Object` if not provided

9.8.2.1 Parameters

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

Field	Contents	Note
var	Variable ∪ DynamicVar	The local variable that will hold this parameter's value
default	ObjectOpt	This parameter's default value; if **none**, this parameter is required

9.8.3 Local Frames

Frames holding bindings for blocks and other statements that can hold local bindings are represented as LocalFrame records (see section 5.11) with the fields below.

Field	Contents	Note
localBindings	LocalBinding{}	Map of qualified names to definitions in this frame
plurality	Plurality	See section 9.8

9.9 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.

9.9.1 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
enumerable	BOOLEAN	**true** if this binding should be visible in a `for-in` statement

A local member is either **forbidden**, a variable, a dynamic variable, a constructor method, a getter, or a setter:

LOCALMEMBER = {**forbidden**} ∪ VARIABLE ∪ DYNAMICVAR ∪ 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	CLASSU	Type of values that may be stored in this variable; **uninitialised** if not determined yet
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
setup	() → CLASSOPT ∪ {**none**, **busy**}	A semantic procedure that performs the Setup action on the variable or constant definition. **none** if the action has already been performed; **busy** if the action is in the process of being performed and should not be reentered.
initialiser	INITIALISER ∪ {**none**, **busy**}	A semantic procedure that computes a variable's initialiser specified by the programmer. **none** if no initialiser was given or if it has already been evaluated; **busy** if the initialiser is being evaluated now and should not be reentered.
initialiserEnv	ENVIRONMENT	The environment to provide to initialiser if this variable is a compile-time constant

The semantic domain VARIABLEOPT consists of all variables as well as **none**:

VARIABLEOPT = VARIABLE ∪ {**none**};

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, **uninitialised** (used when the variable has not been initialised yet and has no default value), or an uninstantiated function (compile time only).

VARIABLEVALUE = OBJECT □ {**uninitialised**} □ UNINSTANTIATEDFUNCTION;

VARIABLEVALUE = OBJECT □ {**inaccessible**, **uninitialised**} □ UNINSTANTIATEDFUNCTION □ () □ OBJECT;

An INITIALISER is a semantic procedure that takes environment and phase parameters and computes a variable's initial value.

INITIALISER = ENVIRONMENT □ PHASE □ OBJECT;

INITIALISEROPT = INITIALISER □ {**none**};

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	ENVIRONMENTU	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	ENVIRONMENTU	The environment bound to this setter

9.9.2 Instance Bindings

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. This record is also used as a key to look up an instance's SLOT (see section 9.1.9.1).

Field	Contents	Note
multiname	MULTINAME	The set of qualified names for this instance variable

final	BOOLEAN	**true** if this instance variable may not be overridden in subclasses
enumerable	BOOLEAN	**true** if this instance variable's `public` name should be visible in a `for-in` statement
type	CLASS	Type of values that may be stored in this variable
defaultValue	OBJECTU	This variable's default value, if provided
immutable	BOOLEAN	**true** if this variable's value may not be changed once set

The semantic domain INSTANCEVARIABLEOPT consists of all instance variables as well as **none**:

INSTANCEVARIABLEOPT = INSTANCEVARIABLE □ {**none**};

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

Field	Contents	Note
multiname	MULTINAME	The set of qualified names for this instance method
final	BOOLEAN	**true** if this instance method may not be overridden in subclasses
enumerable	BOOLEAN	**true** if this instance method's `public` name should be visible in a `for-in` statement
signature	PARAMETERFRAME	This method's signature
call	OBJECT □ OBJECT[] □ ENVIRONMENT □ PHASE □ OBJECT	A procedure to call when this instance method is invoked. The procedure takes a `this` OBJECT, a list of argument OBJECTs, an ENVIRONMENT from the `env` field below, and a PHASE (see section 9.5) and produces an OBJECT result
env	ENVIRONMENT	The environment to pass to call

The semantic domain INSTANCEMETHODOPT consists of all instance methods as well as **none**:

INSTANCEMETHODOPT = INSTANCEMETHOD □ {**none**};

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

Field	Contents	Note
multiname	MULTINAME	The set of qualified names for this getter
final	BOOLEAN	**true** if this getter may not be overridden in subclasses
enumerable	BOOLEAN	**true** if this getter's `public` name should be visible in a `for-in` statement
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 to pass to call

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

Field	Contents	Note
multiname	MULTINAME	The set of qualified names for this setter
final	BOOLEAN	**true** if this setter may not be overridden in subclasses
enumerable	BOOLEAN	**true** if this setter's `public` name should be visible in a `for-in` statement
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 `this` value, the value being written, the environment from the env field below, and the current mode of expression evaluation
env	ENVIRONMENTI	The environment to pass to call

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): $\{0 \ldots 2^{32} – 1\}$
  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): $\{–2^{31} \ldots 2^{31} – 1\}$
  j: INTEGER ← *bitwiseAnd*(*i*, 0xFFFFFFFF);
  if $j \geq 2^{31}$ **then** $j \leftarrow j – 2^{32}$ **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): $\{0 \ldots 2^{64} – 1\}$
  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): $\{–2^{63} \ldots 2^{63} – 1\}$
  j: INTEGER ← *bitwiseAnd*(*i*, 0xFFFFFFFFFFFFFFFF);
  if $j \geq 2^{63}$ **then** $j \leftarrow j – 2^{64}$ **end if**;
  return *j*
end proc;

proc *truncateToInteger*(*x*: GENERALNUMBER): INTEGER
  case *x* **of**
    $\{+\infty_{f32}, +\infty_{f64}, –\infty_{f32}, –\infty_{f64}, \mathrm{NaN}_{f32}, \mathrm{NaN}_{f64}\}$ **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**
    {**NaN**$_{f32}$, **NaN**$_{f64}$, **+∞**$_{f32}$, **+∞**$_{f64}$, **–∞**$_{f32}$, **–∞**$_{f64}$} **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** i_{long}
  elsif $2^{63} \le i \le 2^{64} - 1$ **then return** i_{ulong}
  else return *realToFloat64*(*i*)
  end if
end proc;

proc *integerToULong*(*i*: INTEGER): GENERALNUMBER
  if $0 \le i \le 2^{64} - 1$ **then return** i_{ulong}
  elsif $-2^{63} \le i \le -1$ **then return** i_{long}
  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** i_{long} **else return** i_{ulong} **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** i_{ulong} **else return** i_{long} **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 ∈ {NaNf32, NaNf64} or y ∈ {NaNf32, NaNf64} 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 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 ≠ 0;
    FLOAT32 **do return** *o* □ {**+zero**$_{f32}$, **–zero**$_{f32}$, **NaN**$_{f32}$};
    FLOAT64 **do return** *o* □ {**+zero**$_{f64}$, **–zero**$_{f64}$, **NaN**$_{f64}$};
    STRING **do return** *o* ≠ “”;
    CHARACTER □ NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ SIMPLEINSTANCE □ METHODCLOSURE □
      DATE □ REGEXP □ PACKAGE **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 NaN**$_{f64}$;
    NULL □ {**false**} **do return +zero**$_{f64}$;
    {**true**} **do return** 1.0$_{f64}$;
    GENERALNUMBER **do return** *o*;
    CHARACTER □ STRING **do** ????;
    NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ METHODCLOSURE □ PACKAGE **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 **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 ≥ 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
      {NaNf32} do return “NaN”;
      {+zerof32, –zerof32} 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 ≥ 1, 10^(k–1) ≤ s ≤ 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 ≤ n ≤ 21 then return digits ⊕ repeat(‘0’, n – k)
            elsif 0 < n ≤ 21 then return digits[0 ... n – 1] ⊕ “.” ⊕ digits[n ...]
            elsif –6 < n ≤ 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
      {NaNf64} do return "NaN";
      {+zerof64, –zerof64} 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 ≥ 1, 10^(k–1) ≤ s ≤ 10^k, realToFloat64(s10^(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 s10^(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 ≤ n ≤ 21 then return digits ⊕ repeat('0', n – k)
            elsif 0 < n ≤ 21 then return digits[0 ... n – 1] ⊕ "." ⊕ digits[n ...]
            elsif –6 < n ≤ 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 *toQualifiedName*

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

```
proc toQualifiedName(o: OBJECT, phase: PHASE): QUALIFIEDNAME
   return public::(toString(o, phase))
end proc;
```

10.2.6 *toPrimitive*

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

10.2.7 *toClass*

```
proc toClass(o: OBJECT): CLASS
   if o  CLASS then return o else throw badValueError end if
end proc;
```

10.2.8 Attributes

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

```
proc combineAttributes(a: AttributeOptNotFalse, b: Attribute): Attribute
   if b = false then return false
   elsif a ∈ {none, true} then return b
   elsif b = true then return a
   elsif a ∈ Namespace then
      if a = b then return a
      elsif b ∈ Namespace then
         return CompoundAttribute⟨⟨namespaces: {a, b}, explicit: false, enumerable: 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, enumerable: a.enumerable or b.enumerable,
               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, enumerable: false, dynamic: false,
               memberMod: none, overrideMod: none, prototype: false, unused: false⟩⟩
      Namespace do
         return CompoundAttribute⟨⟨namespaces: {a}, explicit: false, enumerable: false, dynamic: false,
               memberMod: none, overrideMod: none, prototype: false, unused: false⟩⟩
      CompoundAttribute do return a
   end case
end proc;
```

10.3 Access Utilities

```
proc selectPrimaryName(multiname: Multiname): QualifiedName
   if |multiname| = 1 then return the one element of multiname
   elsif some qname ∈ multiname satisfies qname.namespace = public then return qname
   else throw propertyAccessError
   end if
end proc;

proc accessesOverlap(accesses1: AccessSet, accesses2: AccessSet): Boolean
   return accesses1 = accesses2 or accesses1 = readWrite or accesses2 = readWrite
end proc;
```

```
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;
```

setupVariable(*v*) runs Setup and initialises the type of the variable *v*, making sure that Setup is done at most once and does not reenter itself.

```
proc setupVariable(v: VARIABLE)
   setup: () → CLASSOPT ∪ {none, busy} ← v.setup;
   case setup of
      () → CLASSOPT do
         v.setup ← busy;
         type: CLASSOPT ← setup();
         if type = none then type ← objectClass end if;
         note Variables cannot be written by compile-time constant expressions, so v.type = uninitialised and
               v.value = uninitialised must still hold.
         v.type ← type;
         v.setup ← none;
      {none} do nothing;
      {busy} do throw propertyAccessError
   end case
end proc;
```

```
proc writeVariable(v: VARIABLE, newValue: OBJECT, clearInitialiser: BOOLEAN): OBJECT
   type: CLASS ← getVariableType(v);
   coercedValue: OBJECT ← type.implicitCoerce(newValue, false);
   if clearInitialiser then v.initialiser ← none end if;
   if v.immutable and (v.value ≠ uninitialised or v.initialiser ≠ none) then
      throw propertyAccessError
   end if;
   v.value ← coercedValue;
   return coercedValue
end proc;
```

```
proc getVariableType(v: VARIABLE): CLASS
   type: CLASSU ← v.type;
   if type = uninitialised then throw propertyAccessError end if;
   return type
end proc;
```

10.4 Environmental Utilities

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 with frame or local block frame, a local block frame directly enclosed in a class, or a local block frame directly enclosed in a with frame directly enclosed in a class.

proc *getRegionalEnvironment*(*env*: ENVIRONMENT): FRAME[]
  i: INTEGER ← 0;
  while *env*[*i*] ∈ LOCALFRAME ∪ WITHFRAME **do** *i* ← *i* + 1 **end while**;
  if *env*[*i*] ∈ CLASS **then while** *i* ≠ 0 **and** *env*[*i*] ∉ LOCALFRAME **do** *i* ← *i* – 1 **end while**
  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 *getPackageFrame*(*env*: ENVIRONMENT): PACKAGE
  g: FRAME ← *env*[|*env*| – 2];
  note The penultimate frame *g* is always a PACKAGE.
  return *g*
end proc;

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): OBJECTUOPT
  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;

10.5 Property Lookup

tag propertyLookup;

tuple LEXICALLOOKUP
  this: OBJECTUOPT
end tuple;

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

proc *findLocalMember*(*o*: NONWITHFRAME ∪ 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 instanceMemberAccesses(m: InstanceMember): AccessSet
   case m of
      InstanceVariable ∪ InstanceMethod do return readWrite;
      InstanceGetter do return read;
      InstanceSetter do return write
   end case
end proc;

proc findLocalInstanceMember(c: Class, multiname: Multiname, accesses: AccessSet): InstanceMemberOpt
   matchingMembers: InstanceMember{} ← {m | ∀m ∈ c.instanceMembers such that
         m.multiname ∩ multiname ≠ {} and accessesOverlap(instanceMemberAccesses(m), accesses)};
   if matchingMembers = {} then return none
   elsif |matchingMembers| = 1 then return the one element of matchingMembers
   else
      note This access is ambiguous because it found several different instance members in the same class.
      throw propertyAccessError
   end if
end proc;

proc findCommonMember(o: Object, multiname: Multiname, access: Access, flat: Boolean):
      {none} ∪ LocalMember ∪ InstanceMember
   m: {none} ∪ LocalMember ∪ InstanceMember;
   case o of
      Undefined ∪ Null ∪ Boolean ∪ Long ∪ ULong ∪ Float32 ∪ Float64 ∪ Character ∪ String ∪
            Namespace ∪ CompoundAttribute ∪ MethodClosure do
         return none;
      SimpleInstance ∪ RegExp ∪ Date ∪ Package do
         m ← findLocalMember(o, multiname, access);
      Class do
         m ← findLocalMember(o, multiname, access);
         if m = none then m ← findLocalInstanceMember(o, multiname, access) end if
   end case;
   if m ≠ none then return m end if;
   super: ObjectOpt ← o.super;
   if super ≠ none then
      m ← findCommonMember(super, multiname, access, flat);
      if flat and m ∈ DynamicVar then m ← none end if
   end if;
   return m
end proc;

proc findBaseInstanceMember(c: Class, multiname: Multiname, accesses: AccessSet): InstanceMemberOpt
   note Start from the root class (Object) and proceed through more specific classes that are ancestors of c.
   for each s ∈ ancestors(c) do
      m: InstanceMemberOpt ← findLocalInstanceMember(s, multiname, accesses);
      if m ≠ none then return m end if
   end for each;
   return none
end proc;
```

getDerivedInstanceMember(*c*, *mBase*, *accesses*) returns the most derived instance member whose name includes that of *mBase* and whose access includes *access*. The caller of *getDerivedInstanceMember* ensures that such a member always exists. If *accesses* is **readWrite** then it is possible that this search could find both a getter and a setter defined in the same class; in this case either the getter or the setter is returned at the implementation's discretion.

```
proc getDerivedInstanceMember(c: Class, mBase: InstanceMember, accesses: AccessSet): InstanceMember
   if some m ∈ c.instanceMembers satisfies mBase.multiname ∩ m.multiname and
         accessesOverlap(instanceMemberAccesses(m), accesses) then
      return m
   else return getDerivedInstanceMember(c.super, mBase, accesses)
   end if
end proc;
```

```
proc lookupInstanceMember(c: Class, qname: QualifiedName, access: Access): InstanceMemberOpt
   mBase: InstanceMemberOpt ← findBaseInstanceMember(c, {qname}, access);
   if mBase = none then return none end if;
   return getDerivedInstanceMember(c, mBase, access)
end proc;
```

10.6 Reading

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

dotRead(*o*, *multiname*, *phase*) is a simplified interface to read the *multiname* property of *o*.

```
proc dotRead(o: Object, multiname: Multiname, phase: Phase): Object
   limit: Class ← objectType(o);
   result: ObjectOpt ← limit.read(o, limit, multiname, propertyLookup, phase);
   if result = none then throw propertyAccessError end if;
   return result
end proc;
```

```
proc defaultBracketRead(o: Object, limit: Class, args: Object[], phase: Phase): ObjectOpt
   if |args| ≠ 1 then throw argumentMismatchError end if;
   qname: QualifiedName ← toQualifiedName(args[0], phase);
   return limit.read(o, limit, {qname}, propertyLookup, phase)
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 ← none;
    case frame of
      PACKAGE ∪ CLASS do
        limit: CLASS ← objectType(frame);
        result ← limit.read(frame, limit, multiname, kind, phase);
      SYSTEMFRAME ∪ PARAMETERFRAME ∪ LOCALFRAME do
        m: LOCALMEMBEROPT ← findLocalMember(frame, multiname, read);
        if m ≠ none then result ← readLocalMember(m, phase) end if;
      WITHFRAME do
        value: OBJECTU ← frame.value;
        if value = uninitialised then
          if phase = compile then throw compileExpressionError
          else throw propertyAccessError
          end if
        end if;
        limit: CLASS ← objectType(value);
        result ← limit.read(value, limit, multiname, kind, phase)
    end case;
    if result ≠ none then return result end if;
    i ← i + 1
  end while;
  throw referenceError
end proc;

proc defaultReadProperty(o: OBJECT, limit: CLASS, multiname: MULTINAME, kind: LOOKUPKIND, phase: PHASE):
    OBJECTOPT
  mBase: INSTANCEMEMBEROPT ← findBaseInstanceMember(limit, multiname, read);
  if mBase ≠ none then return readInstanceMember(o, limit, mBase, phase) end if;
  if limit ≠ objectType(o) then return none end if;
  m: {none} ∪ LOCALMEMBER ∪ INSTANCEMEMBER ← findCommonMember(o, multiname, read, false);
  case m of
    {none} do
      if kind = propertyLookup and o ∈ SIMPLEINSTANCE ∪ DATE ∪ REGEXP ∪ PACKAGE and not o.sealed 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);
    INSTANCEMEMBER do
      if o ∈ CLASS or kind = propertyLookup then throw propertyAccessError end if;
      this: OBJECTUOPT ← kind.this;
      case this of
        {none} do throw propertyAccessError;
        {uninitialised} do throw compileExpressionError;
        OBJECT do return readInstanceMember(this, objectType(this), m, phase)
      end case
  end case
end proc;
```

readInstanceProperty(*o*, *qname*, *phase*) is a simplified interface to *defaultReadProperty* used to read to instance members that are known to exist.

proc *readInstanceProperty*(*o*: OBJECT, *qname*: QUALIFIEDNAME, *phase*: PHASE): OBJECT
  c: CLASS *objectType*(*o*);
  mBase: INSTANCEMEMBEROPT *findBaseInstanceMember*(*c*, {*qname*}, **read**);
  note *readInstanceProperty* is only called in cases where the instance property is known to exist, so *mBase* cannot be **none**here.
  return *readInstanceMember*(*o*, *c*, *mBase*, *phase*)
end proc;

proc *readInstanceMember*(*this*: OBJECT, *c*: CLASS, *mBase*: INSTANCEMEMBER, *phase*: PHASE): OBJECT
  m: INSTANCEMEMBER *getDerivedInstanceMember*(*c*, *mBase*, **read**);
  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** **propertyAccessError** **end if**;
      return *v*;
    INSTANCEMETHOD **do return** METHODCLOSUREthis: *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;
      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;
      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;
            {uninitialised} do
               if not m.immutable then throw propertyAccessError end if;
               note Try to run a const variable's initialiser if there is one.
               setupVariable(m);
               initialiser: Initialiser □ {none, busy} □ m.initialiser;
               if initialiser □ {none, busy} then throw propertyAccessError end if;
               m.initialiser □ busy;
               coercedValue: Object;
               try
                  newValue: Object □ initialiser(m.initialiserEnv, compile);
                  coercedValue □ writeVariable(m, newValue, true)
               catch x: SemanticException do
                  note If initialisation failed, restore m.initialiser to its original value so it can be tried later.
                  m.initialiser □ initialiser;
                  throw x
               end try;
               return coercedValue;
            UninstantiatedFunction do
               note An uninstantiated function can only be found when phase = compile.
               throw compileExpressionError
         end case;
      ConstructorMethod do return m.code;
      Getter do
         env: EnvironmentU □ m.env;
         if env = uninitialised 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;
```

10.7 Writing

If *r* is a reference, *writeReference*(*r*, *newValue*) writes *newValue* into *r*. An error occurs if *r* is not a reference. *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);
         result ← ok;
      DotReference do
         result ← r.limit.write(r.base, r.limit, r.propertyMultiname, propertyLookup, true, newValue, phase);
      BracketReference do
         result ← r.limit.bracketWrite(r.base, r.limit, r.args, newValue, phase)
   end case;
   if result = none then throw propertyAccessError end if
end proc;
```

dotWrite(*o*, *multiname*, *newValue*, *phase*) is a simplified interface to write *newValue* into the *multiname* property of *o*.

```
proc dotWrite(o: Object, multiname: Multiname, newValue: Object, phase: {run})
   limit: Class ← objectType(o);
   result: {none, ok} ← limit.write(o, limit, multiname, propertyLookup, true, newValue, phase);
   if result = none then throw propertyAccessError end if
end proc;

proc indexWrite(o: Object, i: Integer, newValue: Object, phase: {run})
   if i < 0 or i ≥ arrayLimit then throw rangeError end if;
   limit: Class ← objectType(o);
   result: {none, ok} ← limit.bracketWrite(o, limit, [i_ulong], newValue, phase);
   if result = none then throw propertyAccessError end if
end proc;

proc defaultBracketWrite(o: Object, limit: Class, args: Object[], newValue: Object, phase: {run}): {none, ok}
   if |args| ≠ 1 then throw argumentMismatchError end if;
   qname: QualifiedName ← toQualifiedName(args[0], phase);
   return limit.write(o, limit, {qname}, propertyLookup, true, newValue, phase)
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} ← none;
      case frame of
         Package ∪ Class do
            limit: Class ← objectType(frame);
            result ← limit.write(frame, limit, multiname, kind, false, newValue, phase);
         SystemFrame ∪ ParameterFrame ∪ LocalFrame do
            m: LocalMemberOpt ← findLocalMember(frame, multiname, write);
            if m ≠ none then writeLocalMember(m, newValue, phase); result ← ok
            end if;
         WithFrame do
            value: ObjectU ← frame.value;
            if value = uninitialised then throw propertyAccessError end if;
            limit: Class ← objectType(value);
            result ← limit.write(value, limit, multiname, kind, false, newValue, phase)
      end case;
      if result = ok then return end if;
      i ← i + 1
   end while;
   if createIfMissing then
      pkg: Package ← getPackageFrame(env);
      note Try to write the variable into pkg again, this time allowing new dynamic bindings to be created dynamically.
      limit: Class ← objectType(pkg);
      result: {none, ok} ← limit.write(pkg, limit, multiname, kind, true, newValue, phase);
      if result = ok then return end if
   end if;
   throw referenceError
end proc;
```

proc *defaultWriteProperty*(*o*: OBJECT, *limit*: CLASS, *multiname*: MULTINAME, *kind*: LOOKUPKIND,
createIfMissing: BOOLEAN, *newValue*: OBJECT, *phase*: {**run**}): {**none**, **ok**}
mBase: INSTANCEMEMBEROPT *findBaseInstanceMember*(*limit*, *multiname*, **write**);
if *mBase* ≠ **none then** *writeInstanceMember*(*o*, *limit*, *mBase*, *newValue*, *phase*); **return ok**
end if;
if *limit* ≠ *objectType*(*o*) **then return none end if**;
m: {**none**} LOCALMEMBER INSTANCEMEMBER *findCommonMember*(*o*, *multiname*, **write**, **true**);
case *m* **of**
{**none**} **do**
if *createIfMissing* **and** *o* SIMPLEINSTANCE DATE REGEXP PACKAGE **and not** *o*.sealed **then**
qname: QUALIFIEDNAME *selectPrimaryName*(*multiname*);
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 *findBaseInstanceMember*(*objectType*(*o*), {*qname*}, **read**) = **none and**
findCommonMember(*o*, {*qname*}, **read**, **true**) = **none then**
createDynamicProperty(*o*, *qname*, **false**, **true**, *newValue*);
return ok
end if
end if;
return none;
LOCALMEMBER **do** *writeLocalMember*(*m*, *newValue*, *phase*); **return ok**;
INSTANCEMEMBER **do**
if *o* CLASS **or** *kind* = **propertyLookup then throw propertyAccessError end if**;
this: OBJECTUOPT *kind*.this;
note *this* cannot be **uninitialised** during the **run** phase.
case *this* **of**
{**none**} **do throw propertyAccessError**;
OBJECT **do**
writeInstanceMember(*this*, *objectType*(*this*), *m*, *newValue*, *phase*);
return ok
end case
end case
end proc;

The caller must make sure that the created property does not already exist and does not conflict with any other property.

proc *createDynamicProperty*(*o*: SIMPLEINSTANCE DATE REGEXP PACKAGE, *qname*: QUALIFIEDNAME,
sealed: BOOLEAN, *enumerable*: BOOLEAN, *newValue*: OBJECT)
dv: DYNAMICVAR **new** DYNAMICVAR value: *newValue*, sealed: *sealed*;
o.localBindings *o*.localBindings {LOCALBINDINGqname: *qname*, accesses: **readWrite**, content: *dv*,
explicit: **false**, enumerable: *enumerable*}
end proc;

```
proc writeInstanceMember(this: OBJECT, c: CLASS, mBase: INSTANCEMEMBER, newValue: OBJECT, phase: {run})
  m: INSTANCEMEMBER ← getDerivedInstanceMember(c, mBase, write);
  case m of
    INSTANCEVARIABLE do
      s: SLOT ← findSlot(this, m);
      coercedValue: OBJECT ← m.type.implicitCoerce(newValue, false);
      if m.immutable and s.value ≠ uninitialised then throw propertyAccessError
      end if;
      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 writeVariable(m, newValue, false);
    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: ENVIRONMENTU ← m.env;
      note All instances are resolved for the run phase, so env ≠ uninitialised.
      m.call(coercedValue, env, phase)
  end case
end proc;
```

10.8 Deleting

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 result ← true end if;
    LEXICALREFERENCE do result ← lexicalDelete(r.env, r.variableMultiname, phase);
    DOTREFERENCE do
      result ← r.limit.delete(r.base, r.limit, r.propertyMultiname, propertyLookup, phase);
    BRACKETREFERENCE do
      result ← r.limit.bracketDelete(r.base, r.limit, r.args, phase)
  end case;
  if result ≠ none then return result else return true end if
end proc;

proc defaultBracketDelete(o: OBJECT, limit: CLASS, args: OBJECT[], phase: {run}): BOOLEANOPT
  if |args| ≠ 1 then throw argumentMismatchError end if;
  qname: QUALIFIEDNAME ← toQualifiedName(args[0], phase);
  return limit.delete(o, limit, {qname}, propertyLookup, phase)
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 ← none;
      case frame of
         PACKAGE ∪ CLASS do
            limit: CLASS ← objectType(frame);
            result ← limit.delete(frame, limit, multiname, kind, phase);
         SYSTEMFRAME ∪ PARAMETERFRAME ∪ LOCALFRAME do
            if findLocalMember(frame, multiname, write) ≠ none then result ← false
            end if;
         WITHFRAME do
            value: OBJECTU ← frame.value;
            if value = uninitialised then throw propertyAccessError end if;
            limit: CLASS ← objectType(value);
            result ← limit.delete(value, limit, multiname, kind, phase)
      end case;
      if result ≠ none then return result end if;
      i ← i + 1
   end while;
   return true
end proc;

proc defaultDeleteProperty(o: OBJECT, limit: CLASS, multiname: MULTINAME, kind: LOOKUPKIND, phase: {run}):
      BOOLEANOPT
   if findBaseInstanceMember(limit, multiname, write) ≠ none then return false end if;
   if limit ≠ objectType(o) then return none end if;
   m: {none} ∪ LOCALMEMBER ∪ INSTANCEMEMBER ← findCommonMember(o, 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
            o.localBindings ← {b | ∀b ∈ o.localBindings such that b.qname ∉ multiname or b.content ≠ m};
            return true
         end if;
      INSTANCEMEMBER do
         if o ∉ CLASS or kind = propertyLookup then return false end if;
         this: OBJECTUOPT ← kind.this;
         note this cannot be uninitialised during the run phase.
         case this of
            {none} do throw propertyAccessError;
            OBJECT do return false
         end case
   end case
end proc;
```

10.9 Enumerating

```
proc defaultEnumerate(o: OBJECT): OBJECT{}
   e1: OBJECT{} ← enumerateInstanceMembers(objectType(o));
   e2: OBJECT{} ← enumerateCommonMembers(o);
   return e1 ∪ e2
end proc;

proc enumerateInstanceMembers(c: CLASS): OBJECT{}
   e: OBJECT{} ← {};
   for each m ∈ c.instanceMembers do
      if m.enumerable then
         e ← e ∪ {qname.id | ∃qname ∈ m.multiname such that qname.namespace = public}
      end if
   end for each;
   super: CLASSOPT ← c.super;
   if super = none then return e else return e ∪ enumerateInstanceMembers(super) end if
end proc;

proc enumerateCommonMembers(o: OBJECT): OBJECT{}
   case o of
      UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHARACTER ∪ STRING ∪
            NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ METHODCLOSURE do
         return {};
      CLASS ∪ SIMPLEINSTANCE ∪ REGEXP ∪ DATE ∪ PACKAGE do
         e: OBJECT{} ← {};
         for each b ∈ o.localBindings do
            if b.enumerable and b.qname.namespace = public then e ← e ∪ {b.qname.id}
            end if
         end for each;
         super: OBJECTOPT ← o.super;
         if super ≠ none then e ← e ∪ enumerateCommonMembers(super) end if;
         return e
   end case
end proc;
```

10.10 Creating Instances

```
proc createSimpleInstance(c: CLASS, super: OBJECTOPT,
      call: OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT ∪ {none},
      construct: SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT ∪ {none}, env: ENVIRONMENTOPT): SIMPLEINSTANCE
   slots: SLOT{} ← {};
   for each s ∈ ancestors(c) do
      for each m ∈ s.instanceMembers do
         if m ∈ INSTANCEVARIABLE then
            slot: SLOT ← new SLOT⟦id: m, value: m.defaultValue⟧;
            slots ← slots ∪ {slot}
         end if
      end for each
   end for each;
   return new SIMPLEINSTANCE⟦localBindings: {}, super: super, sealed: not c.dynamic, type: c, slots: slots,
         call: call, construct: construct, env: env⟧
end proc;
```

10.11 Adding Local Definitions

proc *defineLocalMember*(*env*: ENVIRONMENT, *id*: STRING, *namespaces*: NAMESPACE{},
overrideMod: OVERRIDEMODIFIER, *explicit*: BOOLEAN, *accesses*: ACCESSSET, *m*: LOCALMEMBER): MULTINAME
innerFrame: NONWITHFRAME ← *env*[0];
if *overrideMod* ≠ **none** **or** (*explicit* **and** *innerFrame* ∉ PACKAGE) **then**
throw **definitionError**
end if;
namespaces2: NAMESPACE{} ← *namespaces*;
if *namespaces2* = {} **then** *namespaces2* ← {*public*} **end if**;
multiname: MULTINAME ← {*ns*::*id* | ∀*ns* ∈ *namespaces2*};
regionalEnv: FRAME[] ← *getRegionalEnvironment*(*env*);
if some *b* ∈ *innerFrame*.localBindings **satisfies**
b.qname ∈ *multiname* **and** *accessesOverlap*(*b*.accesses, *accesses*) **then**
throw **definitionError**
end if;
for each *frame* ∈ *regionalEnv*[1 ...] **do**
if *frame* ∉ WITHFRAME **and** (**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*, enumerable: **true**⟩ | ∀*qname* ∈ *multiname*};
innerFrame.localBindings ← *innerFrame*.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**, enumerable: **true**⟩ | ∀*qname* ∈ *multiname*};
for each *frame* ∈ *regionalEnv*[1 ...] **do**
if *frame* ∉ WITHFRAME **then**
frame.localBindings ← *frame*.localBindings ∪ *newForbiddenBindings*
end if
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 package 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 ← public::id;
    regionalEnv: Frame[] ← getRegionalEnvironment(env);
    regionalFrame: Frame ← regionalEnv[|regionalEnv| – 1];
    note env is either a Package or a ParameterFrame because hoisting only occurs into package 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, enumerable: true⟩⟩};
        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 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;
        m.sealed ← true;
        regionalFrame.localBindings ← regionalFrame.localBindings – {b};
        regionalFrame.localBindings ← regionalFrame.localBindings ∪
                {LocalBinding⟨⟨enumerable: true, other fields from b⟩⟩};
        return m
    end if
end proc;
```

10.12 Adding Instance Definitions

```
proc searchForOverrides(c: Class, multiname: Multiname, accesses: AccessSet): InstanceMemberOpt
    mBase: InstanceMemberOpt ← none;
    s: ClassOpt ← c.super;
    if s ≠ none then
        for each qname ∈ multiname do
            m: InstanceMemberOpt ← findBaseInstanceMember(s, {qname}, accesses);
            if mBase = none then mBase ← m
            elsif m ≠ none and m ≠ mBase then throw definitionError
            end if
        end for each
    end if;
    return mBase
end proc;
```

```
proc defineInstanceMember(c: Class, cxt: Context, id: String, namespaces: Namespace{},
      overrideMod: OverrideModifier, explicit: Boolean, m: InstanceMember): InstanceMemberOpt
   if explicit then throw definitionError end if;
   accesses: AccessSet ← instanceMemberAccesses(m);
   requestedMultiname: Multiname ← {ns::id | ∀ns ∈ namespaces};
   openMultiname: Multiname ← {ns::id | ∀ns ∈ cxt.openNamespaces};
   definedMultiname: Multiname;
   searchedMultiname: Multiname;
   if requestedMultiname = {} then
      definedMultiname ← {public::id};
      searchedMultiname ← openMultiname;
      note definedMultiname ⊆ searchedMultiname because the public namespace is always open.
   else definedMultiname ← requestedMultiname; searchedMultiname ← requestedMultiname
   end if;
   mBase: InstanceMemberOpt ← searchForOverrides(c, searchedMultiname, accesses);
   mOverridden: InstanceMemberOpt ← none;
   if mBase ≠ none then
      mOverridden ← getDerivedInstanceMember(c, mBase, accesses);
      definedMultiname ← mOverridden.multiname;
      if not (requestedMultiname ⊆ definedMultiname) then throw definitionError
      end if;
      goodKind: Boolean;
      case m of
         InstanceVariable do goodKind ← mOverridden ∈ InstanceVariable;
         InstanceGetter do
            goodKind ← mOverridden ∈ InstanceVariable ∪ InstanceGetter;
         InstanceSetter do
            goodKind ← mOverridden ∈ InstanceVariable ∪ InstanceSetter;
         InstanceMethod do goodKind ← mOverridden ∈ InstanceMethod
      end case;
      if mOverridden.final or not goodKind then throw definitionError end if
   end if;
   if some m2 ∈ c.instanceMembers satisfies m2.multiname ∩ definedMultiname ≠ {} and
         accessesOverlap(instanceMemberAccesses(m2), accesses) then
      throw definitionError
   end if;
   case overrideMod of
      {none} do
         if mBase ≠ none or searchForOverrides(c, openMultiname, accesses) ≠ none then
            throw definitionError
         end if;
      {false} do if mBase ≠ none then throw definitionError end if;
      {true} do if mBase = none then throw definitionError end if;
      {undefined} do nothing
   end case;
   m.multiname ← definedMultiname;
   c.instanceMembers ← c.instanceMembers ∪ {m};
   return mOverridden
end proc;
```

10.13 Instantiation

proc *instantiateFunction*(*uf*: UNINSTANTIATEDFUNCTION, *env*: ENVIRONMENT): SIMPLEINSTANCE
    c: CLASS ← *uf*.type;
    i: SIMPLEINSTANCE ← *createSimpleInstance*(*c*, *c*.prototype, *uf*.call, *uf*.construct, *env*);
    dotWrite(*i*, {*public*::“`length`”}, *realToFloat64*(*uf*.length), **run**);
    if *uf*.buildPrototype **then**
        prototype: OBJECT ← *prototypeClass*.construct([], **run**);
        dotWrite(*prototype*, {*public*::“`constructor`”}, *i*, **run**);
        dotWrite(*i*, {*public*::“`prototype`”}, *prototype*, **run**)
    end if;
    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
         note m.setup = none because Setup must have been called on a frame before that frame can be instantiated.
         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, setup: none,
               initialiser: m.initialiser, initialiserEnv: env□□
      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;
            {uninitialised} do
               return new Getter□□type: m.type, call: m.call, env: env□□
         end case;
      Setter do
         case m.env of
            Environment do return m;
            {uninitialised} 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 instantiateLocalFrame(pluralFrame: LocalFrame, env: Environment): LocalFrame
   singularFrame: LocalFrame □ new LocalFrame□□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, returnType: *pluralFrame*.returnType⟩⟩;

note *pluralMembers* will contain the set of all LOCALMEMBER records found in the *pluralFrame*.

pluralMembers: LOCALMEMBER{} ← {*b*.content | ∀*b* ∈ *pluralFrame*.localBindings};

note If any of the parameters (including the rest parameter) are anonymous, their bindings will not be present in *pluralFrame*.localBindings. In this situation, the following steps add their LOCALMEMBER records to *pluralMembers*.

for each *p* ∈ *pluralFrame*.parameters **do** *pluralMembers* ← *pluralMembers* ∪ {*p*.var}

end for each;

rest: VARIABLEOPT ← *pluralFrame*.rest;

if *rest* ≠ **none then** *pluralMembers* ← *pluralMembers* ∪ {*rest*} **end if**;

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.parameters ← [PARAMETER⟨⟨var: *translateMember*(*op*.var), default: *op*.default⟩⟩ | ∀*op* ∈ *pluralFrame*.parameters];

if *rest* = **none then** *singularFrame*.rest ← **none**

else *singularFrame*.rest ← *translateMember*(*rest*)

end if;

return *singularFrame*

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 β:

β ∈ {allowIn, noIn}

Most expression productions have both the Validate and Eval actions defined. Most of the Eval actions on subexpressions produce an OBJORREF result, indicating that the subexpression may evaluate to either a value or a place that can potentially be read, written, or deleted (see section 9.3).

12.1 Identifiers

An *Identifier* is either a non-keyword **Identifier** token or one of the non-reserved keywords `get`, `set`, `exclude`, 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**

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`";

12.2 Qualified Identifiers

Syntax

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

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

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

QualifiedIdentifier ⇒
  SimpleQualifiedIdentifier
| *ExpressionQualifiedIdentifier*

Validation

OpenNamespaces[*Qualifier*]: NAMESPACE{};

proc Validate[*Qualifier*] (*cxt*: Context, *env*: Environment)
[*Qualifier* → *Identifier*] **do** OpenNamespaces[*Qualifier*] ← *cxt*.openNamespaces;
[*Qualifier* → `public`] **do nothing**;
[*Qualifier* → `private`] **do**
c: ClassOpt ← *getEnclosingClass*(*env*);
if *c* = **none then throw syntaxError end if**
end proc;

OpenNamespaces[*SimpleQualifiedIdentifier*]: Namespace{};

proc Validate[*SimpleQualifiedIdentifier*] (*cxt*: Context, *env*: Environment)
[*SimpleQualifiedIdentifier* → *Identifier*] **do**
OpenNamespaces[*SimpleQualifiedIdentifier*] ← *cxt*.openNamespaces;
[*SimpleQualifiedIdentifier* → *Qualifier* `::` *Identifier*] **do**
Validate[*Qualifier*](*cxt*, *env*)
end proc;

proc Validate[*ExpressionQualifiedIdentifier* → *ParenExpression* `::` *Identifier*] (*cxt*: Context, *env*: Environment)
Validate[*ParenExpression*](*cxt*, *env*)
end proc;

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

Setup

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

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

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

Evaluation

proc Eval[*Qualifier*] (*env*: Environment, *phase*: Phase): Namespace
[*Qualifier* → *Identifier*] **do**
multiname: Multiname ← {*ns*::(Name[*Identifier*]) | ∀*ns* ∈ OpenNamespaces[*Qualifier*]};
a: Object ← *lexicalRead*(*env*, *multiname*, *phase*);
if *a* ∉ Namespace **then throw badValueError end if**;
return *a*;
[*Qualifier* → `public`] **do return** *public*;
[*Qualifier* → `private`] **do**
c: ClassOpt ← *getEnclosingClass*(*env*);
note Validate already ensured that *c* ≠ **none**.
return *c*.privateNamespace
end proc;

proc Eval[*SimpleQualifiedIdentifier*] (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME
  [*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do**
    return {*ns*::(Name[*Identifier*]) | ∀*ns* ∈ OpenNamespaces[*SimpleQualifiedIdentifier*]};
  [*SimpleQualifiedIdentifier* ⇒ *Qualifier* **::** *Identifier*] **do**
    q: NAMESPACE ← Eval[*Qualifier*](*env*, *phase*);
    return {*q*::(Name[*Identifier*])}
end proc;

proc Eval[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **::** *Identifier*]
    (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME
  q: OBJECT ← *readReference*(Eval[*ParenExpression*](*env*, *phase*), *phase*);
  if *q* ∉ NAMESPACE **then throw** **badValueError** **end if**;
  return {*q*::(Name[*Identifier*])}
end proc;

proc Eval[*QualifiedIdentifier*] (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME
  [*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] **do**
    return Eval[*SimpleQualifiedIdentifier*](*env*, *phase*);
  [*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] **do**
    return Eval[*ExpressionQualifiedIdentifier*](*env*, *phase*)
end proc;

12.3 Primary Expressions

Syntax

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

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

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

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** *public*;
[*PrimaryExpression* □ **Number**] **do return** Value[**Number**];
[*PrimaryExpression* □ **String**] **do return** Value[**String**];
[*PrimaryExpression* □ `this`] **do**
this: OBJECTUOPT □ *findThis*(*env*, **true**);
note Validate ensured that *this* ≠ **none** at this point.
if *this* = **uninitialised 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, uninitialised} □ unchecked ? uninitialised : none;
    localCxt: CONTEXT □ new CONTEXT□□strict: cxt.strict, openNamespaces: cxt.openNamespaces,
        constructsSuper: none□□;
    F[FunctionExpression] □ ValidateStaticFunction[FunctionCommon](localCxt, env, this, unchecked,
        unchecked);
```

[*FunctionExpression* ⇒ **function** *Identifier FunctionCommon*] **do**
v: VARIABLE ← **new** VARIABLE⟨⟨type: *functionClass*, value: **uninitialised**, immutable: **true**, setup: **none**, initialiser: **busy**, initialiserEnv: *env*⟩⟩;
b: LOCALBINDING ← LOCALBINDING⟨qname: *public*::(Name[*Identifier*]), accesses: **readWrite**, content: *v*, explicit: **false**, enumerable: **true**⟩;
compileFrame: LOCALFRAME ← **new** LOCALFRAME⟨⟨localBindings: {*b*}, plurality: **plural**⟩⟩;
unchecked: BOOLEAN ← **not** *cxt*.strict **and** Plain[*FunctionCommon*];
this: {**none**, **uninitialised**} ← *unchecked* ? **uninitialised** : **none**;
localCxt: CONTEXT ← **new** CONTEXT⟨⟨strict: *cxt*.strict, openNamespaces: *cxt*.openNamespaces, constructsSuper: **none**⟩⟩;
F[*FunctionExpression*] ← ValidateStaticFunction[*FunctionCommon*](*localCxt*, [*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: *functionClass*, value: **uninitialised**, immutable: **true**, setup: **none**, initialiser: **none**, initialiserEnv: *env*⟩⟩;
b: LOCALBINDING ← LOCALBINDING⟨qname: *public*::(Name[*Identifier*]), accesses: **readWrite**, content: *v*, explicit: **false**, enumerable: **true**⟩;
runtimeFrame: LOCALFRAME ← **new** LOCALFRAME⟨⟨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 ⇒
«empty»
| *NonemptyFieldList*

NonemptyFieldList ⇒
LiteralField
| *LiteralField* **,** *NonemptyFieldList*

LiteralField ⇒ *FieldName* **:** *AssignmentExpression*^allowIn

FieldName ⇒
  QualifiedIdentifier
| **String**
| **Number**
| *ParenExpression*

Validation

proc Validate[*ObjectLiteral* ⇒ **{** *FieldList* **}**] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*FieldList*](*cxt*, *env*)
end proc;

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

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

proc Validate[*LiteralField* ⇒ *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*FieldName*](*cxt*, *env*);
  Validate[*AssignmentExpression*$^{\text{allowIn}}$](*cxt*, *env*)
end proc;

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

Setup

proc Setup[*ObjectLiteral* ⇒ **{** *FieldList* **}**] ()
  Setup[*FieldList*]()
end proc;

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

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

proc Setup[*LiteralField* ⇒ *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$] ()
  Setup[*FieldName*]();
  Setup[*AssignmentExpression*$^{\text{allowIn}}$]()
end proc;

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

Evaluation

proc Eval[*ObjectLiteral* ⇒ **{** *FieldList* **}**] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
  if *phase* = **compile then throw compileExpressionError end if**;
  o: OBJECT ← *prototypeClass*.construct([], *phase*);
  Eval[*FieldList*](*env*, *o*, *phase*);
  return *o*
end proc;

Eval[*FieldList*] (*env*: ENVIRONMENT, *o*: OBJECT, *phase*: {**run**}) propagates the call to Eval to every nonterminal in the expansion of *FieldList*.

Eval[*NonemptyFieldList*] (*env*: ENVIRONMENT, *o*: OBJECT, *phase*: {**run**}) propagates the call to Eval to every nonterminal in the expansion of *NonemptyFieldList*.

proc Eval[*LiteralField* → *FieldName* **:** *AssignmentExpression*$^{\text{allowIn}}$] (*env*: ENVIRONMENT, *o*: OBJECT, *phase*: {**run**})
  multiname: MULTINAME → Eval[*FieldName*](*env*, *phase*);
  value: OBJECT → *readReference*(Eval[*AssignmentExpression*$^{\text{allowIn}}$](*env*, *phase*), *phase*);
  dotWrite(*o*, *multiname*, *value*, *phase*)
end proc;

proc Eval[*FieldName*] (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME
  [*FieldName* → *QualifiedIdentifier*] **do return** Eval[*QualifiedIdentifier*](*env*, *phase*);
  [*FieldName* → **String**] **do return** {*toQualifiedName*(Value[**String**], *phase*)};
  [*FieldName* → **Number**] **do return** {*toQualifiedName*(Value[**Number**], *phase*)};
  [*FieldName* → *ParenExpression*] **do**
    a: OBJECT → *readReference*(Eval[*ParenExpression*](*env*, *phase*), *phase*);
    return {*toQualifiedName*(*a*, *phase*)}
end proc;

12.6 Array Literals

Syntax

ArrayLiteral → **[** *ElementList* **]**

ElementList →
  «empty»
  | *LiteralElement*
  | **,** *ElementList*
  | *LiteralElement* **,** *ElementList*

LiteralElement → *AssignmentExpression*$^{\text{allowIn}}$

Validation

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

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

proc Validate[*LiteralElement* → *AssignmentExpression*$^{\text{allowIn}}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*AssignmentExpression*$^{\text{allowIn}}$](*cxt*, *env*)
end proc;

Setup

proc Setup[*ArrayLiteral* → **[** *ElementList* **]**] ()
  Setup[*ElementList*]()
end proc;

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

```
proc Setup[LiteralElement ⇒ AssignmentExpression^allowIn] ()
   Setup[AssignmentExpression^allowIn]()
end proc;
```

Evaluation

```
proc Eval[ArrayLiteral ⇒ [ ElementList ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   if phase = compile then throw compileExpressionError end if;
   o: OBJECT ← arrayClass.construct([], phase);
   length: INTEGER ← Eval[ElementList](env, 0, o, phase);
   if length > arrayLimit then throw rangeError end if;
   dotWrite(o, {arrayPrivate::"length"}, length_ulong, phase);
   return o
end proc;

proc Eval[ElementList] (env: ENVIRONMENT, length: INTEGER, o: OBJECT, phase: {run}): INTEGER
   [ElementList ⇒ «empty»] do return length;
   [ElementList ⇒ LiteralElement] do
      Eval[LiteralElement](env, length, o, phase);
      return length + 1;
   [ElementList0 ⇒ , ElementList1] do
      return Eval[ElementList1](env, length + 1, o, phase);
   [ElementList0 ⇒ LiteralElement , ElementList1] do
      Eval[LiteralElement](env, length, o, phase);
      return Eval[ElementList1](env, length + 1, o, phase)
end proc;

proc Eval[LiteralElement ⇒ AssignmentExpression^allowIn]
      (env: ENVIRONMENT, length: INTEGER, o: OBJECT, phase: {run})
   value: OBJECT ← readReference(Eval[AssignmentExpression^allowIn](env, phase), phase);
   indexWrite(o, length, value, phase)
end proc;
```

12.7 Super Expressions

Syntax

```
SuperExpression ⇒
     super
   | super ParenExpression
```

Validation

```
proc Validate[SuperExpression] (cxt: CONTEXT, env: ENVIRONMENT)
   [SuperExpression ⇒ super] do
      c: CLASSOPT ← getEnclosingClass(env);
      if c = none or findThis(env, false) = none then throw syntaxError end if;
      if c.super = none then throw definitionError 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: OBJECTUOPT ← *findThis*(*env*, **false**);
    note Validate ensured that *this* cannot be **none** at this point.
    if *this* = **uninitialised then throw compileExpressionError end if**;
    return *makeLimitedInstance*(*this*, *getEnclosingClass*(*env*), *phase*);
  [*SuperExpression* ⇒ **super** *ParenExpression*] **do**
    r: OBJORREF ← Eval[*ParenExpression*](*env*, *phase*);
    return *makeLimitedInstance*(*r*, *getEnclosingClass*(*env*), *phase*)
end proc;

proc *makeLimitedInstance*(*r*: OBJORREF, *c*: CLASS, *phase*: PHASE): OBJOPTIONALLIMIT
  o: OBJECT ← *readReference*(*r*, *phase*);
  limit: CLASSOPT ← *c*.super;
  note Validate ensured that *limit* cannot be **none** at this point.
  coerced: OBJECT ← *limit*.implicitCoerce(*o*, **false**);
  if *coerced* = **null then return null end if**;
  return LIMITEDINSTANCE⟨⟨instance: *coerced*, 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**
  m: MULTINAME ← Eval[*SimpleQualifiedIdentifier*](*env*, *phase*);
  return LEXICALREFERENCE⟨env: *env*, variableMultiname: *m*, 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;
  case *r* **of**
    OBJECT ∪ LEXICALREFERENCE **do** *base* ← **null**;
    DOTREFERENCE ∪ BRACKETREFERENCE **do** *base* ← *r*.base
  end case;
  args: OBJECT[] ← 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**
  m: MULTINAME ← Eval[*ExpressionQualifiedIdentifier*](*env*, *phase*);
  return LEXICALREFERENCE⟨env: *env*, variableMultiname: *m*, 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;
  case *r* **of**
    OBJECT ∪ LEXICALREFERENCE **do** *base* ← **null**;
    DOTREFERENCE ∪ BRACKETREFERENCE **do** *base* ← *r*.base
  end case;
  args: OBJECT[] ← 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: Object[] ← 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
        m: Multiname ← Eval[QualifiedIdentifier](env, phase);
        return LexicalReference⟨⟨env: env, variableMultiname: m, 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, [], 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;
```

proc *call*(*this*: OBJECT, *a*: OBJECT, *args*: OBJECT[], *phase*: PHASE): OBJECT
  case *a* **of**
    UNDEFINED ∪ NULL ∪ BOOLEAN ∪ GENERALNUMBER ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪
      COMPOUNDATTRIBUTE ∪ DATE ∪ REGEXP ∪ PACKAGE **do**
      throw badValueError;
    CLASS **do return** *a*.call(*this*, *args*, *phase*);
    SIMPLEINSTANCE **do**
      f: OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT ∪ {**none**} ← *a*.call;
      if *f* = **none then throw badValueError end if**;
      return *f*(*this*, *a*, *args*, *phase*);
    METHODCLOSURE **do**
      m: INSTANCEMETHOD ← *a*.method;
      return *m*.call(*a*.this, *args*, *m*.env, *phase*)
  end case
end proc;

proc *construct*(*a*: OBJECT, *args*: OBJECT[], *phase*: PHASE): OBJECT
  case *a* **of**
    UNDEFINED ∪ NULL ∪ BOOLEAN ∪ GENERALNUMBER ∪ CHARACTER ∪ STRING ∪ NAMESPACE ∪
      COMPOUNDATTRIBUTE ∪ METHODCLOSURE ∪ DATE ∪ REGEXP ∪ PACKAGE **do**
      throw badValueError;
    CLASS **do return** *a*.construct(*args*, *phase*);
    SIMPLEINSTANCE **do**
      f: SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT ∪ {**none**} ← *a*.construct;
      if *f* = **none then throw badValueError end if**;
      return *f*(*a*, *args*, *phase*)
  end case
end proc;

12.9 Member Operators

Syntax

MemberOperator ⇒
  . *QualifiedIdentifier*
  | *Brackets*

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

Arguments ⇒
  ( )
  | *ParenListExpression*

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*$^{\text{allowIn}}$]] **do** Validate[*ListExpression*$^{\text{allowIn}}$](*cxt*, *env*)
end proc;

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

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*.

Evaluation

```
proc Eval[MemberOperator] (env: ENVIRONMENT, base: OBJOPTIONALLIMIT, phase: PHASE): OBJORREF
   [MemberOperator  . QualifiedIdentifier] do
      m: MULTINAME  Eval[QualifiedIdentifier](env, phase);
      case base of
         OBJECT do
            return DOTREFERENCEbase: base, limit: objectType(base), propertyMultiname: m
         LIMITEDINSTANCE do
            return DOTREFERENCEbase: base.instance, limit: base.limit, propertyMultiname: m
      end case;
   [MemberOperator  Brackets] do
      args: OBJECT[]  Eval[Brackets](env, phase);
      case base of
         OBJECT do
            return BRACKETREFERENCEbase: base, limit: objectType(base), args: args
         LIMITEDINSTANCE do
            return BRACKETREFERENCEbase: base.instance, limit: base.limit, args: args
      end case
end proc;

proc Eval[Brackets] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
   [Brackets  [ ]] do return [];
   [Brackets  [ ListExpression^allowIn ]] do
      return EvalAsList[ListExpression^allowIn](env, phase)
end proc;

proc Eval[Arguments] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
   [Arguments  ( )] do return [];
   [Arguments  ParenListExpression] do
      return EvalAsList[ParenListExpression](env, phase)
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*);

```
    [UnaryExpression₀ ⇒ void UnaryExpression₁] do
        readReference(Eval[UnaryExpression₁](env, phase), phase);
        return undefined;
    [UnaryExpression₀ ⇒ typeof UnaryExpression₁] do
        a: OBJECT ← readReference(Eval[UnaryExpression₁](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, 1.0f64, 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, 1.0f64, phase);
        writeReference(r, c, phase);
        return c;
    [UnaryExpression₀ ⇒ + UnaryExpression₁] do
        a: OBJECT ← readReference(Eval[UnaryExpression₁](env, phase), phase);
        return plus(a, phase);
    [UnaryExpression₀ ⇒ - UnaryExpression₁] do
        a: OBJECT ← readReference(Eval[UnaryExpression₁](env, phase), phase);
        return minus(a, phase);
    [UnaryExpression ⇒ - NegatedMinLong] do return (–2^63)long;
    [UnaryExpression₀ ⇒ ~ UnaryExpression₁] do
        a: OBJECT ← readReference(Eval[UnaryExpression₁](env, phase), phase);
        return bitNot(a, phase);
    [UnaryExpression₀ ⇒ ! UnaryExpression₁] do
        a: OBJECT ← readReference(Eval[UnaryExpression₁](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*: $\{-2^{63} \ldots 2^{63} - 1\}$ ← *x*.value; **return** *bitwiseXor*(*i*, –1)$_{\text{long}}$;
    ULONG **do**
      i: $\{0 \ldots 2^{64} - 1\}$ ← *x*.value;
      return *bitwiseXor*(*i*, 0xFFFFFFFFFFFFFFFF)$_{\text{ulong}}$;
    FLOAT32 ∪ FLOAT64 **do**
      i: $\{-2^{31} \ldots 2^{31} - 1\}$ ← *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*);

```
  [MultiplicativeExpression0 ⇒ MultiplicativeExpression1 * UnaryExpression] do
    a: OBJECT ← readReference(Eval[MultiplicativeExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[UnaryExpression](env, phase), phase);
    return multiply(a, b, phase);
  [MultiplicativeExpression0 ⇒ MultiplicativeExpression1 / UnaryExpression] do
    a: OBJECT ← readReference(Eval[MultiplicativeExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[UnaryExpression](env, phase), phase);
    return divide(a, b, phase);
  [MultiplicativeExpression0 ⇒ MultiplicativeExpression1 % UnaryExpression] do
    a: OBJECT ← readReference(Eval[MultiplicativeExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[UnaryExpression](env, phase), phase);
    return remainder(a, b, phase)
end proc;

proc multiply(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← toGeneralNumber(a, phase);
  y: GENERALNUMBER ← toGeneralNumber(b, phase);
  if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
    i: 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 float64Multiply(toFloat64(x), toFloat64(y))
end proc;

proc divide(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← toGeneralNumber(a, phase);
  y: GENERALNUMBER ← toGeneralNumber(b, phase);
  if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
    i: INTEGEROPT ← checkInteger(x);
    j: INTEGEROPT ← checkInteger(y);
    if i ≠ none and j ≠ none and j ≠ 0 then
      q: RATIONAL ← i/j;
      if x ∈ ULONG or y ∈ 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 ← *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 and** *j* ≠ 0 **then**
      q: RATIONAL ← *i*/*j*;
      k: INTEGER ← $q \geq 0$? ⌊*q*⌋ : ⌈*q*⌉;
      r: INTEGER ← *i* – *j*⋅*k*;
      if *x* ∈ ULONG **or** *y* ∈ 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*) ⊕ *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* ≠ **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 *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 i_long;
      ULONG do
         count ← bitwiseAnd(count, 0x3F);
         i: {0 ... 2^64 – 1} ← unsignedWrap64(bitwiseShift(x.value, count));
         return i_ulong
   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: $\{-2^{31} \ldots 2^{31} - 1\}$ *signedWrap32*(*truncateToInteger*(*x*));
      count *bitwiseAnd*(*count*, 0x1F);
      i *bitwiseShift*(*i*, –*count*);
      return *realToFloat64*(*i*);
    LONG **do**
      count *bitwiseAnd*(*count*, 0x3F);
      i: $\{-2^{63} \ldots 2^{63} - 1\}$ *bitwiseShift*(*x*.value, –*count*);
      return $i_{\textbf{long}}$;
    ULONG **do**
      count *bitwiseAnd*(*count*, 0x3F);
      i: $\{-2^{63} \ldots 2^{63} - 1\}$ *bitwiseShift*(*signedWrap64*(*x*.value), –*count*);
      return $(\textit{unsignedWrap64}(i))_{\textbf{ulong}}$
  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: $\{0 \ldots 2^{32} - 1\}$ *unsignedWrap32*(*truncateToInteger*(*x*));
      count *bitwiseAnd*(*count*, 0x1F);
      i *bitwiseShift*(*i*, –*count*);
      return *realToFloat64*(*i*);
    LONG **do**
      count *bitwiseAnd*(*count*, 0x3F);
      i: $\{0 \ldots 2^{64} - 1\}$ *bitwiseShift*(*unsignedWrap64*(*x*.value), –*count*);
      return $(\textit{signedWrap64}(i))_{\textbf{long}}$;
    ULONG **do**
      count *bitwiseAnd*(*count*, 0x3F);
      i: $\{0 \ldots 2^{64} - 1\}$ *bitwiseShift*(*x*.value, –*count*);
      return $i_{\textbf{ulong}}$
  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$^{\mathrm{noIn}}$ □
    ShiftExpression
  | *RelationalExpression*$^{\mathrm{noIn}}$ **<** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **>** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **<=** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **>=** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **is** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **as** *ShiftExpression*
  | *RelationalExpression*$^{\mathrm{noIn}}$ **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*□$_0$ □ *RelationalExpression*□$_1$ **is** *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*.is(*a*);
  [*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*$^{\text{allowIn}}{}_0$ □ *RelationalExpression*$^{\text{allowIn}}{}_1$ **in** *ShiftExpression*] **do**
a: OBJECT □ *readReference*(Eval[*RelationalExpression*$^{\text{allowIn}}{}_1$](*env*, *phase*), *phase*);
b: OBJECT □ *readReference*(Eval[*ShiftExpression*](*env*, *phase*), *phase*);
qname: QUALIFIEDNAME □ *toQualifiedName*(*a*, *phase*);
c: CLASS □ *objectType*(*b*);
return *findBaseInstanceMember*(*c*, {*qname*}, **read**) ≠ **none** **or**
findBaseInstanceMember(*c*, {*qname*}, **write**) ≠ **none** **or**
findCommonMember(*b*, {*qname*}, **read**, **false**) ≠ **none** **or**
findCommonMember(*b*, {*qname*}, **write**, **false**) ≠ **none**;
[*RelationalExpression*$^{\square}$ □ *RelationalExpression*$^{\square}$ **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$^{\square}$ □
RelationalExpression$^{\square}$
| *EqualityExpression*$^{\square}$ **==** *RelationalExpression*$^{\square}$
| *EqualityExpression*$^{\square}$ **!=** *RelationalExpression*$^{\square}$
| *EqualityExpression*$^{\square}$ **===** *RelationalExpression*$^{\square}$
| *EqualityExpression*$^{\square}$ **!==** *RelationalExpression*$^{\square}$

Validation

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

Setup

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

Evaluation

proc Eval[*EqualityExpression*$^{\square}$] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*EqualityExpression*$^{\square}$ □ *RelationalExpression*$^{\square}$] **do**
return Eval[*RelationalExpression*$^{\square}$](*env*, *phase*);

```
  [EqualityExpressionβ0 ⇒ EqualityExpressionβ1 == RelationalExpressionβ] do
    a: Object ← readReference(Eval[EqualityExpressionβ1](env, phase), phase);
    b: Object ← readReference(Eval[RelationalExpressionβ](env, phase), phase);
    return isEqual(a, b, phase);
  [EqualityExpressionβ0 ⇒ EqualityExpressionβ1 != RelationalExpressionβ] do
    a: Object ← readReference(Eval[EqualityExpressionβ1](env, phase), phase);
    b: Object ← readReference(Eval[RelationalExpressionβ](env, phase), phase);
    return not isEqual(a, b, phase);
  [EqualityExpressionβ0 ⇒ EqualityExpressionβ1 === RelationalExpressionβ] do
    a: Object ← readReference(Eval[EqualityExpressionβ1](env, phase), phase);
    b: Object ← readReference(Eval[RelationalExpressionβ](env, phase), phase);
    return isStrictlyEqual(a, b, phase);
  [EqualityExpressionβ0 ⇒ EqualityExpressionβ1 !== RelationalExpressionβ] do
    a: Object ← readReference(Eval[EqualityExpressionβ1](env, phase), phase);
    b: Object ← readReference(Eval[RelationalExpressionβ](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 do
      case b of
        Undefined ∪ Null do return false;
        Namespace ∪ CompoundAttribute ∪ Class ∪ MethodClosure ∪ SimpleInstance ∪ Date ∪
            RegExp ∪ Package 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$^{\beta}$ →
  EqualityExpression$^{\beta}$
  | *BitwiseAndExpression*$^{\beta}$ **&** *EqualityExpression*$^{\beta}$

BitwiseXorExpression$^{\beta}$ →
  BitwiseAndExpression$^{\beta}$
  | *BitwiseXorExpression*$^{\beta}$ **^** *BitwiseAndExpression*$^{\beta}$

BitwiseOrExpression$^{\beta}$ →
  BitwiseXorExpression$^{\beta}$
  | *BitwiseOrExpression*$^{\beta}$ **|** *BitwiseXorExpression*$^{\beta}$

Validation

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

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

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

Setup

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

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

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

Evaluation

proc Eval[*BitwiseAndExpression*$^{\beta}$] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
  [*BitwiseAndExpression*$^{\beta}$ → *EqualityExpression*$^{\beta}$] **do**
    return Eval[*EqualityExpression*$^{\beta}$](*env*, *phase*);
  [*BitwiseAndExpression*$^{\beta}_0$ → *BitwiseAndExpression*$^{\beta}_1$ **&** *EqualityExpression*$^{\beta}$] **do**
    a: OBJECT ← *readReference*(Eval[*BitwiseAndExpression*$^{\beta}_1$](*env*, *phase*), *phase*);
    b: OBJECT ← *readReference*(Eval[*EqualityExpression*$^{\beta}$](*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 ... 2^63 – 1} □ signedWrap64(truncateToInteger(x));
      j: {–2^63 ... 2^63 – 1} □ signedWrap64(truncateToInteger(y));
      k: {–2^63 ... 2^63 – 1} □ bitwiseAnd(i, j);
      if x □ ULONG or y □ ULONG then return (unsignedWrap64(k))ulong
      else return k_long
      end if
   else
      i: {–2^31 ... 2^31 – 1} □ signedWrap32(truncateToInteger(x));
      j: {–2^31 ... 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 ... 2^63 – 1} □ signedWrap64(truncateToInteger(x));
      j: {–2^63 ... 2^63 – 1} □ signedWrap64(truncateToInteger(y));
      k: {–2^63 ... 2^63 – 1} □ bitwiseXor(i, j);
      if x □ ULONG or y □ ULONG then return (unsignedWrap64(k))ulong
      else return k_long
      end if
   else
      i: {–2^31 ... 2^31 – 1} □ signedWrap32(truncateToInteger(x));
      j: {–2^31 ... 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** (*unsignedWrap64*(*k*))$_{\mathbf{ulong}}$
        else return $k_{\mathbf{long}}$
        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*$^{\beta}$] () propagates the call to Setup to every nonterminal in the expansion of *ConditionalExpression*$^{\beta}$.

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

Evaluation

proc Eval[*ConditionalExpression*$^{\beta}$] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
  [*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$] **do**
    return Eval[*LogicalOrExpression*$^{\beta}$](*env*, *phase*);
  [*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$ **?** *AssignmentExpression*$^{\beta}_1$ **:** *AssignmentExpression*$^{\beta}_2$] **do**
    a: OBJECT ← *readReference*(Eval[*LogicalOrExpression*$^{\beta}$](*env*, *phase*), *phase*);
    if *toBoolean*(*a*, *phase*) **then**
      return *readReference*(Eval[*AssignmentExpression*$^{\beta}_1$](*env*, *phase*), *phase*)
    else return *readReference*(Eval[*AssignmentExpression*$^{\beta}_2$](*env*, *phase*), *phase*)
    end if
end proc;

proc Eval[*NonAssignmentExpression*$^{\beta}$] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
  [*NonAssignmentExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$] **do**
    return Eval[*LogicalOrExpression*$^{\beta}$](*env*, *phase*);
  [*NonAssignmentExpression*$^{\beta}_0$ ⇒ *LogicalOrExpression*$^{\beta}$ **?** *NonAssignmentExpression*$^{\beta}_1$ **:** *NonAssignmentExpression*$^{\beta}_2$] **do**
    a: OBJECT ← *readReference*(Eval[*LogicalOrExpression*$^{\beta}$](*env*, *phase*), *phase*);
    if *toBoolean*(*a*, *phase*) **then**
      return *readReference*(Eval[*NonAssignmentExpression*$^{\beta}_1$](*env*, *phase*), *phase*)
    else return *readReference*(Eval[*NonAssignmentExpression*$^{\beta}_2$](*env*, *phase*), *phase*)
    end if
end proc;

12.19 Assignment Operators

Syntax

AssignmentExpression$^{\beta}$ ⇒
  ConditionalExpression$^{\beta}$
  | *PostfixExpression* **=** *AssignmentExpression*$^{\beta}$
  | *PostfixExpression CompoundAssignment AssignmentExpression*$^{\beta}$
  | *PostfixExpression LogicalAssignment AssignmentExpression*$^{\beta}$

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

LogicalAssignment □
    &&=
  | **^^=**
  | **||=**

Semantics

tag **andEq**;

tag **xorEq**;

tag **orEq**;

Validation

proc Validate[*AssignmentExpression*□] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  [*AssignmentExpression*□ □ *ConditionalExpression*□] **do**
    Validate[*ConditionalExpression*□](*cxt*, *env*);
  [*AssignmentExpression*□$_0$ □ *PostfixExpression* **=** *AssignmentExpression*□$_1$] **do**
    Validate[*PostfixExpression*](*cxt*, *env*);
    Validate[*AssignmentExpression*□$_1$](*cxt*, *env*);
  [*AssignmentExpression*□$_0$ □ *PostfixExpression CompoundAssignment AssignmentExpression*□$_1$] **do**
    Validate[*PostfixExpression*](*cxt*, *env*);
    Validate[*AssignmentExpression*□$_1$](*cxt*, *env*);
  [*AssignmentExpression*□$_0$ □ *PostfixExpression LogicalAssignment AssignmentExpression*□$_1$] **do**
    Validate[*PostfixExpression*](*cxt*, *env*);
    Validate[*AssignmentExpression*□$_1$](*cxt*, *env*)
end proc;

Setup

proc Setup[*AssignmentExpression*□] ()
  [*AssignmentExpression*□ □ *ConditionalExpression*□] **do** Setup[*ConditionalExpression*□]();
  [*AssignmentExpression*□$_0$ □ *PostfixExpression* **=** *AssignmentExpression*□$_1$] **do**
    Setup[*PostfixExpression*]();
    Setup[*AssignmentExpression*□$_1$]();
  [*AssignmentExpression*□$_0$ □ *PostfixExpression CompoundAssignment AssignmentExpression*□$_1$] **do**
    Setup[*PostfixExpression*]();
    Setup[*AssignmentExpression*□$_1$]();
  [*AssignmentExpression*□$_0$ □ *PostfixExpression LogicalAssignment AssignmentExpression*□$_1$] **do**
    Setup[*PostfixExpression*]();
    Setup[*AssignmentExpression*□$_1$]()
end proc;

Evaluation

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

```
    [AssignmentExpression^β_0 ⇒ PostfixExpression = AssignmentExpression^β_1] do
      if phase = compile then throw compileExpressionError end if;
      ra: OBJORREF ← Eval[PostfixExpression](env, phase);
      b: OBJECT ← readReference(Eval[AssignmentExpression^β_1](env, phase), phase);
      writeReference(ra, b, phase);
      return b;
    [AssignmentExpression^β_0 ⇒ PostfixExpression CompoundAssignment AssignmentExpression^β_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^β_1](env, phase), phase);
      result: OBJECT ← Op[CompoundAssignment](oLeft, oRight, phase);
      writeReference(rLeft, result, phase);
      return result;
    [AssignmentExpression^β_0 ⇒ PostfixExpression LogicalAssignment AssignmentExpression^β_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^β_1](env, phase), phase)
          end if;
        {xorEq} do
          bRight: BOOLEAN ← toBoolean(readReference(Eval[AssignmentExpression^β_1](env, phase), phase), phase);
          result ← bLeft xor bRight;
        {orEq} do
          if not bLeft then
            result ← readReference(Eval[AssignmentExpression^β_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*□

Validation

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

Setup

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

Evaluation

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

proc EvalAsList[*ListExpression*□] (*env*: ENVIRONMENT, *phase*: PHASE): OBJECT[]
  [*ListExpression*□ □ *AssignmentExpression*□] **do**
    elt: OBJECT □ *readReference*(Eval[*AssignmentExpression*□](*env*, *phase*), *phase*);
    return **[***elt***]**;
  [*ListExpression*□$_0$ □ *ListExpression*□$_1$ `,` *AssignmentExpression*□] **do**
    elts: OBJECT[] □ EvalAsList[*ListExpression*□$_1$](*env*, *phase*);
    elt: OBJECT □ *readReference*(Eval[*AssignmentExpression*□](*env*, *phase*), *phase*);
    return *elts* ⊕ **[***elt***]**
end proc;

12.21 Type Expressions

Syntax

TypeExpression□ □ *NonAssignmentExpression*□

Validation

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

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

$\omega \in$ {abbrev, noShortIf, full}

Statement$^\omega$ ⇒
   ExpressionStatement Semicolon$^\omega$
 | *SuperStatement Semicolon*$^\omega$
 | *Block*
 | *LabeledStatement*$^\omega$
 | *IfStatement*$^\omega$
 | *SwitchStatement*
 | *DoStatement Semicolon*$^\omega$
 | *WhileStatement*$^\omega$
 | *ForStatement*$^\omega$
 | *WithStatement*$^\omega$
 | *ContinueStatement Semicolon*$^\omega$
 | *BreakStatement Semicolon*$^\omega$
 | *ReturnStatement Semicolon*$^\omega$
 | *ThrowStatement Semicolon*$^\omega$
 | *TryStatement*

Substatement$^\omega$ ⇒
   EmptyStatement
 | *Statement*$^\omega$
 | *SimpleVariableDefinition Semicolon*$^\omega$
 | *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*$^{\text{full}}$] **do**
    Validate[*SubstatementsPrefix*$_1$](*cxt*, *env*, *jt*);
    Validate[*Substatement*$^{\text{full}}$](*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*$^{\text{abbrev}}$ □ «empty»] **do nothing**;
  [*Semicolon*$^{\text{noShortIf}}$ □ «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*□ □ *Attributes* [no line break] **{** *Substatements* **}**] **do**
if Enabled[*Substatement*□] **then return** Eval[*Substatements*](*env*, *d*)
else return *d*
end if
end proc;

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

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

13.1 Empty Statement

Syntax

EmptyStatement □ **;**

13.2 Expression Statement

Syntax

ExpressionStatement □ [lookahead□{**function**, **{**}] *ListExpression*$^{\text{allowIn}}$

Validation

proc Validate[*ExpressionStatement* □ [lookahead□{**function**, **{**}] *ListExpression*$^{\text{allowIn}}$]
(*cxt*: CONTEXT, *env*: ENVIRONMENT)
Validate[*ListExpression*$^{\text{allowIn}}$](*cxt*, *env*)
end proc;

Setup

proc Setup[*ExpressionStatement* ⇒ [lookahead∉{**function**, **{**}] *ListExpression*$^{\text{allowIn}}$] ()
  Setup[*ListExpression*$^{\text{allowIn}}$]()
end proc;

Evaluation

proc Eval[*ExpressionStatement* ⇒ [lookahead∉{**function**, **{**}] *ListExpression*$^{\text{allowIn}}$] (*env*: ENVIRONMENT): OBJECT
  return *readReference*(Eval[*ListExpression*$^{\text{allowIn}}$](*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*]: LOCALFRAME;

proc ValidateUsingFrame[*Block* ⇒ **{** *Directives* **}**]
    (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS, *pl*: PLURALITY, *frame*: FRAME)
  localCxt: CONTEXT ← **new** CONTEXT⟨⟨strict: *cxt*.strict, openNamespaces: *cxt*.openNamespaces,
      constructsSuper: *cxt*.constructsSuper⟩⟩;
  Validate[*Directives*](*localCxt*, [*frame*] ⊕ *env*, *jt*, *pl*, **none**);
  cxt.constructsSuper ← *localCxt*.constructsSuper
end proc;

proc Validate[*Block* ⇒ **{** *Directives* **}**] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS, *pl*: PLURALITY)
compileFrame: LOCALFRAME ← **new** LOCALFRAME⟨⟨localBindings: {}, plurality: *pl*⟩⟩;
CompileFrame[*Block*] ← *compileFrame*;
ValidateUsingFrame[*Block*](*cxt*, *env*, *jt*, *pl*, *compileFrame*)
end proc;

Setup

proc Setup[*Block* ⇒ **{** *Directives* **}**] ()
Setup[*Directives*]()
end proc;

Evaluation

proc Eval[*Block* ⇒ **{** *Directives* **}**] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
compileFrame: LOCALFRAME ← CompileFrame[*Block*];
runtimeFrame: LOCALFRAME;
case *compileFrame*.plurality **of**
{**singular**} **do** *runtimeFrame* ← *compileFrame*;
{**plural**} **do** *runtimeFrame* ← *instantiateLocalFrame*(*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$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$

Validation

proc Validate[*LabeledStatement*$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$]
(*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*$^{\omega}$](*cxt*, *env*, *sl* ∪ {*name*}, *jt2*)
end proc;

Setup

proc Setup[*LabeledStatement*$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$] ()
Setup[*Substatement*$^{\omega}$]()
end proc;

Evaluation

proc Eval[*LabeledStatement*$^{\omega}$ → *Identifier* **:** *Substatement*$^{\omega}$] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
  try return Eval[*Substatement*$^{\omega}$](*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$^{\text{abbrev}}$ →
  if *ParenListExpression* *Substatement*$^{\text{abbrev}}$
 | **if** *ParenListExpression* *Substatement*$^{\text{noShortIf}}$ **else** *Substatement*$^{\text{abbrev}}$

IfStatement$^{\text{full}}$ →
  if *ParenListExpression* *Substatement*$^{\text{full}}$
 | **if** *ParenListExpression* *Substatement*$^{\text{noShortIf}}$ **else** *Substatement*$^{\text{full}}$

IfStatement$^{\text{noShortIf}}$ → **if** *ParenListExpression* *Substatement*$^{\text{noShortIf}}$ **else** *Substatement*$^{\text{noShortIf}}$

Validation

proc Validate[*IfStatement*$^{\omega}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS)
  [*IfStatement*$^{\text{abbrev}}$ → **if** *ParenListExpression* *Substatement*$^{\text{abbrev}}$] **do**
    Validate[*ParenListExpression*](*cxt*, *env*);
    Validate[*Substatement*$^{\text{abbrev}}$](*cxt*, *env*, {}, *jt*);
  [*IfStatement*$^{\text{full}}$ → **if** *ParenListExpression* *Substatement*$^{\text{full}}$] **do**
    Validate[*ParenListExpression*](*cxt*, *env*);
    Validate[*Substatement*$^{\text{full}}$](*cxt*, *env*, {}, *jt*);
  [*IfStatement*$^{\omega}$ → **if** *ParenListExpression* *Substatement*$^{\text{noShortIf}}_1$ **else** *Substatement*$^{\omega}_2$] **do**
    Validate[*ParenListExpression*](*cxt*, *env*);
    Validate[*Substatement*$^{\text{noShortIf}}_1$](*cxt*, *env*, {}, *jt*);
    Validate[*Substatement*$^{\omega}_2$](*cxt*, *env*, {}, *jt*)
end proc;

Setup

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

Evaluation

proc Eval[*IfStatement*$^{\omega}$] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
  [*IfStatement*$^{\text{abbrev}}$ → **if** *ParenListExpression* *Substatement*$^{\text{abbrev}}$] **do**
    o: OBJECT ← *readReference*(Eval[*ParenListExpression*](*env*, **run**), **run**);
    if *toBoolean*(*o*, **run**) **then return** Eval[*Substatement*$^{\text{abbrev}}$](*env*, *d*)
    else return *d*
    end if;

[*IfStatement*$^{\text{full}}$ □ **if** *ParenListExpression Substatement*$^{\text{full}}$] **do**
o: OBJECT □ *readReference*(Eval[*ParenListExpression*](*env*, **run**), **run**);
if *toBoolean*(*o*, **run**) **then return** Eval[*Substatement*$^{\text{full}}$](*env*, *d*)
else return *d*
end if;
[*IfStatement*$^{\square}$ □ **if** *ParenListExpression Substatement*$^{\text{noShortIf}}_{1}$ **else** *Substatement*$^{\square}_{2}$] **do**
o: OBJECT □ *readReference*(Eval[*ParenListExpression*](*env*, **run**), **run**);
if *toBoolean*(*o*, **run**) **then return** Eval[*Substatement*$^{\text{noShortIf}}_{1}$](*env*, *d*)
else return Eval[*Substatement*$^{\square}_{2}$](*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*$^{\text{abbrev}}$

CaseElementsPrefix □
«empty»
| *CaseElementsPrefix CaseElement*$^{\text{full}}$

CaseElement$^{\square}$ □
Directive$^{\square}$
| *CaseLabel*

CaseLabel □
case *ListExpression*$^{\text{allowIn}}$ **:**
| **default :**

Validation

CompileFrame[*SwitchStatement*]: LOCALFRAME;

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: LOCALFRAME ← **new** LOCALFRAME⟨⟨localBindings: {}, plurality: **plural**⟩⟩;
CompileFrame[*SwitchStatement*] ← *compileFrame*;
localCxt: CONTEXT ← **new** CONTEXT⟨⟨strict: *cxt*.strict, openNamespaces: *cxt*.openNamespaces,
constructsSuper: *cxt*.constructsSuper⟩⟩;
Validate[*CaseElements*](*localCxt*, [*compileFrame*] ⊕ *env*, *jt2*);
cxt.constructsSuper ← *localCxt*.constructsSuper
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[$CaseElement^{abbrev}$];

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

NDefaults[*CaseElementsPrefix*]: INTEGER;
NDefaults[*CaseElementsPrefix* ⇒ «empty»] = 0;
NDefaults[$CaseElementsPrefix_0$ ⇒ $CaseElementsPrefix_1$ $CaseElement^{full}$]
= NDefaults[$CaseElementsPrefix_1$] + NDefaults[$CaseElement^{full}$];

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

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

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

proc Validate[*CaseLabel*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS)
[*CaseLabel* ⇒ **case** $ListExpression^{allowIn}$ **:**] **do**
Validate[$ListExpression^{allowIn}$](*cxt*, *env*);
[*CaseLabel* ⇒ **default :**] **do nothing**
end proc;

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: LOCALFRAME ← CompileFrame[*SwitchStatement*];
runtimeFrame: LOCALFRAME ← *instantiateLocalFrame*(*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*$^{\text{abbrev}}$] **do**
guard2: SWITCHGUARD ← Eval[*CaseLabel*](*env*, *guard*, *d*);
guard3: SWITCHGUARD ← Eval[*CaseElementsPrefix*](*env*, *guard2*, *d*);
return Eval[*CaseElement*$^{\text{abbrev}}$](*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*$^{\text{full}}$] **do**
guard2: SWITCHGUARD ← Eval[*CaseElementsPrefix*$_1$](*env*, *guard*, *d*);
return Eval[*CaseElement*$^{\text{full}}$](*env*, *guard2*, *d*)
end proc;

proc Eval[*CaseElement*$^{\omega}$] (*env*: ENVIRONMENT, *guard*: SWITCHGUARD, *d*: OBJECT): SWITCHGUARD
[*CaseElement*$^{\omega}$ → *Directive*$^{\omega}$] **do**
case *guard* **of**
SWITCHKEY ∪ {**default**} **do return** *guard*;
OBJECT **do return** Eval[*Directive*$^{\omega}$](*env*, *guard*)
end case;
[*CaseElement*$^{\omega}$ → *CaseLabel*] **do return** Eval[*CaseLabel*](*env*, *guard*, *d*)
end proc;

proc Eval[*CaseLabel*] (*env*: ENVIRONMENT, *guard*: SWITCHGUARD, *d*: OBJECT): SWITCHGUARD
[*CaseLabel* ⇒ **case** *ListExpression*$^{\text{allowIn}}$ **:**] **do**
case *guard* **of**
{**default**} ∪ OBJECT **do return** *guard*;
SWITCHKEY **do**
label: OBJECT ← *readReference*(Eval[*ListExpression*$^{\text{allowIn}}$](*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*$^{\text{abbrev}}$ **while** *ParenListExpression*

Validation

Labels[*DoStatement*]: LABEL{};

proc Validate[*DoStatement* ⇒ **do** *Substatement*$^{\text{abbrev}}$ **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*$^{\text{abbrev}}$](*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^ω ⇒ **while** ParenListExpression Substatement^ω

Validation

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

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

Setup

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

Evaluation

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

13.10 For Statements

Syntax

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

ForInitialiser
«empty»
| *ListExpression*^noIn
| *VariableDefinition*^noIn
| *Attributes* [no line break] *VariableDefinition*^noIn

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

OptionalExpression
ListExpression^allowIn
| «empty»

Validation

Labels[*ForStatement*]: LABEL{};

CompileLocalFrame[*ForStatement*]: LOCALFRAME;

proc Validate[*ForStatement*^ω] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *sl*: LABEL{}, *jt*: JUMPTARGETS)
[*ForStatement*^ω → **for (** *ForInitialiser* **;** *OptionalExpression*$_1$ **;** *OptionalExpression*$_2$ **)** *Substatement*^ω] **do**
continueLabels: LABEL{} ← *sl* ∪ {**default**};
Labels[*ForStatement*^ω] ← *continueLabels*;
jt2: JUMPTARGETS ← JUMPTARGETS⟦breakTargets: *jt*.breakTargets ∪ {**default**},
continueTargets: *jt*.continueTargets ∪ *continueLabels*⟧;
compileLocalFrame: LOCALFRAME ← **new** LOCALFRAME⟦localBindings: {}, plurality: **plural**⟧;
CompileLocalFrame[*ForStatement*^ω] ← *compileLocalFrame*;
compileEnv: ENVIRONMENT ← [*compileLocalFrame*] ⊕ *env*;
Validate[*ForInitialiser*](*cxt*, *compileEnv*);
Validate[*OptionalExpression*$_1$](*cxt*, *compileEnv*);
Validate[*OptionalExpression*$_2$](*cxt*, *compileEnv*);
Validate[*Substatement*^ω](*cxt*, *compileEnv*, {}, *jt2*);
[*ForStatement*^ω → **for (** *ForInBinding* **in** *ListExpression*^allowIn **)** *Substatement*^ω] **do**
continueLabels: LABEL{} ← *sl* ∪ {**default**};
Labels[*ForStatement*^ω] ← *continueLabels*;
jt2: JUMPTARGETS ← JUMPTARGETS⟦breakTargets: *jt*.breakTargets ∪ {**default**},
continueTargets: *jt*.continueTargets ∪ *continueLabels*⟧;
Validate[*ListExpression*^allowIn](*cxt*, *env*);
compileLocalFrame: LOCALFRAME ← **new** LOCALFRAME⟦localBindings: {}, plurality: **plural**⟧;
CompileLocalFrame[*ForStatement*^ω] ← *compileLocalFrame*;
compileEnv: ENVIRONMENT ← [*compileLocalFrame*] ⊕ *env*;
Validate[*ForInBinding*](*cxt*, *compileEnv*);
Validate[*Substatement*^ω](*cxt*, *compileEnv*, {}, *jt2*)
end proc;

Enabled[*ForInitialiser*]: BOOLEAN;

proc Validate[*ForInitialiser*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ForInitialiser* → «empty»] **do nothing**;
[*ForInitialiser* → *ListExpression*^noIn] **do** Validate[*ListExpression*^noIn](*cxt*, *env*);
[*ForInitialiser* → *VariableDefinition*^noIn] **do**
Validate[*VariableDefinition*^noIn](*cxt*, *env*, **none**);
[*ForInitialiser* → *Attributes* [no line break] *VariableDefinition*^noIn] **do**
Validate[*Attributes*](*cxt*, *env*);
Setup[*Attributes*]();
attr: ATTRIBUTE ← Eval[*Attributes*](*env*, **compile**);
Enabled[*ForInitialiser*] ← *attr* ≠ **false**;
if *attr* ≠ **false then** Validate[*VariableDefinition*^noIn](*cxt*, *env*, *attr*) **end if**
end proc;

proc Validate[*ForInBinding*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
[*ForInBinding* → *PostfixExpression*] **do** Validate[*PostfixExpression*](*cxt*, *env*);
[*ForInBinding* → *VariableDefinitionKind* *VariableBinding*^noIn] **do**
Validate[*VariableBinding*^noIn](*cxt*, *env*, **none**, Immutable[*VariableDefinitionKind*], **true**);

[*ForInBinding* ⇒ *Attributes* [no line break] *VariableDefinitionKind* *VariableBinding*$^{\text{noln}}$] **do**
  Validate[*Attributes*](*cxt*, *env*);
  Setup[*Attributes*]();
  attr: ATTRIBUTE ← Eval[*Attributes*](*env*, **compile**);
  if *attr* = **false then throw definitionError end if**;
  Validate[*VariableBinding*$^{\text{noln}}$](*cxt*, *env*, *attr*, Immutable[*VariableDefinitionKind*], **true**)
end proc;

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

Setup

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

proc Setup[*ForInitialiser*] ()
  [*ForInitialiser* ⇒ «empty»] **do nothing**;
  [*ForInitialiser* ⇒ *ListExpression*$^{\text{noln}}$] **do** Setup[*ListExpression*$^{\text{noln}}$]();
  [*ForInitialiser* ⇒ *VariableDefinition*$^{\text{noln}}$] **do** Setup[*VariableDefinition*$^{\text{noln}}$]();
  [*ForInitialiser* ⇒ *Attributes* [no line break] *VariableDefinition*$^{\text{noln}}$] **do**
    if Enabled[*ForInitialiser*] **then** Setup[*VariableDefinition*$^{\text{noln}}$]() **end if**
end proc;

proc Setup[*ForInBinding*] ()
  [*ForInBinding* ⇒ *PostfixExpression*] **do** Setup[*PostfixExpression*]();
  [*ForInBinding* ⇒ *VariableDefinitionKind* *VariableBinding*$^{\text{noln}}$] **do**
    Setup[*VariableBinding*$^{\text{noln}}$]();
  [*ForInBinding* ⇒ *Attributes* [no line break] *VariableDefinitionKind* *VariableBinding*$^{\text{noln}}$] **do**
    Setup[*VariableBinding*$^{\text{noln}}$]()
end proc;

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

Evaluation

```
proc Eval[ForStatement^Ω] (env: Environment, d: Object): Object
   [ForStatement^Ω ⇒ for ( ForInitialiser ; OptionalExpression1 ; OptionalExpression2 ) Substatement^Ω] do
      runtimeLocalFrame: LocalFrame ← instantiateLocalFrame(CompileLocalFrame[ForStatement^Ω], env);
      runtimeEnv: Environment ← [runtimeLocalFrame] ⊕ env;
      try
         Eval[ForInitialiser](runtimeEnv);
         d1: Object ← d;
         while toBoolean(readReference(Eval[OptionalExpression1](runtimeEnv, run), run), run) do
            try d1 ← Eval[Substatement^Ω](runtimeEnv, d1)
            catch x: SemanticException do
               if x ∈ Continue and x.label ∈ Labels[ForStatement^Ω] then
                  d1 ← x.value
               else throw x
               end if
            end try;
            readReference(Eval[OptionalExpression2](runtimeEnv, run), run)
         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;
```

```
    [ForStatement^ω ⇒ for ( ForInBinding in ListExpression^allowIn ) Substatement^ω] do
        try
            o: OBJECT ← readReference(Eval[ListExpression^allowIn](env, run), run);
            c: CLASS ← objectType(o);
            oldIndices: OBJECT{} ← c.enumerate(o);
            remainingIndices: OBJECT{} ← oldIndices;
            d1: OBJECT ← d;
            while remainingIndices ≠ {} do
                runtimeLocalFrame: LOCALFRAME ← instantiateLocalFrame(CompileLocalFrame[ForStatement^ω], env);
                runtimeEnv: ENVIRONMENT ← [runtimeLocalFrame] ⊕ env;
                index: OBJECT ← any element of remainingIndices;
                remainingIndices ← remainingIndices – {index};
                WriteBinding[ForInBinding](runtimeEnv, index);
                try d1 ← Eval[Substatement^ω](runtimeEnv, d1)
                catch x: SEMANTICEXCEPTION do
                    if x ∈ CONTINUE and x.label ∈ Labels[ForStatement^ω] then
                        d1 ← x.value
                    else throw x
                    end if
                end try;
                newIndices: OBJECT{} ← c.enumerate(o);
                if newIndices ≠ oldIndices then
                    The implementation may, at its discretion, add none, some, or all of the objects in the set difference
                            newIndices – oldIndices to remainingIndices;
                    The implementation may, at its discretion, remove none, some, or all of the objects in the set difference
                            oldIndices – newIndices from remainingIndices;
                end if;
                oldIndices ← newIndices
            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;

proc Eval[ForInitialiser] (env: ENVIRONMENT)
    [ForInitialiser ⇒ «empty»] do nothing;
    [ForInitialiser ⇒ ListExpression^noIn] do
        readReference(Eval[ListExpression^noIn](env, run), run);
    [ForInitialiser ⇒ VariableDefinition^noIn] do
        Eval[VariableDefinition^noIn](env, undefined);
    [ForInitialiser ⇒ Attributes [no line break] VariableDefinition^noIn] do
        if Enabled[ForInitialiser] then Eval[VariableDefinition^noIn](env, undefined)
        end if
end proc;

proc WriteBinding[ForInBinding] (env: ENVIRONMENT, newValue: OBJECT)
    [ForInBinding ⇒ PostfixExpression] do
        r: OBJORREF ← Eval[PostfixExpression](env, run);
        writeReference(r, newValue, run);
    [ForInBinding ⇒ VariableDefinitionKind VariableBinding^noIn] do
        WriteBinding[VariableBinding^noIn](env, newValue);
```

[*ForInBinding* ⇒ *Attributes* [no line break] *VariableDefinitionKind VariableBinding*^noIn] **do**
WriteBinding[*VariableBinding*^noIn](*env*, *newValue*)
end proc;

proc Eval[*OptionalExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF
[*OptionalExpression* ⇒ *ListExpression*^allowIn] **do**
return Eval[*ListExpression*^allowIn](*env*, *phase*);
[*OptionalExpression* ⇒ «empty»] **do return true**
end proc;

13.11 With Statement

Syntax

WithStatement^ω ⇒ **with** *ParenListExpression Substatement*^ω

Validation

CompileLocalFrame[*WithStatement*^ω]: LOCALFRAME;

proc Validate[*WithStatement*^ω ⇒ **with** *ParenListExpression Substatement*^ω]
(*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS)
Validate[*ParenListExpression*](*cxt*, *env*);
compileWithFrame: WITHFRAME ← **new** WITHFRAME⟨⟨value: **uninitialised**⟩⟩;
compileLocalFrame: LOCALFRAME ← **new** LOCALFRAME⟨⟨localBindings: {}, plurality: **plural**⟩⟩;
CompileLocalFrame[*WithStatement*^ω] ← *compileLocalFrame*;
compileEnv: ENVIRONMENT ← [*compileLocalFrame*] ⊕ [*compileWithFrame*] ⊕ *env*;
Validate[*Substatement*^ω](*cxt*, *compileEnv*, {}, *jt*)
end proc;

Setup

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

Evaluation

proc Eval[*WithStatement*^ω ⇒ **with** *ParenListExpression Substatement*^ω] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
value: OBJECT ← *readReference*(Eval[*ParenListExpression*](*env*, **run**), **run**);
runtimeWithFrame: WITHFRAME ← **new** WITHFRAME⟨⟨value: *value*⟩⟩;
runtimeLocalFrame: LOCALFRAME ←
instantiateLocalFrame(CompileLocalFrame[*WithStatement*^ω], [*runtimeWithFrame*] ⊕ *env*);
runtimeEnv: ENVIRONMENT ← [*runtimeLocalFrame*] ⊕ [*runtimeWithFrame*] ⊕ *env*;
return Eval[*Substatement*^ω](*runtimeEnv*, *d*)
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** CONTINUEvalue: *d*, label: **default**
  [*ContinueStatement* **continue** [no line break] *Identifier*] **do**
    throw CONTINUEvalue: *d*, label: Name[*Identifier*]
end proc;

proc Eval[*BreakStatement*] (*env*: ENVIRONMENT, *d*: OBJECT): OBJECT
  [*BreakStatement* **break**] **do throw** BREAKvalue: *d*, label: **default**
  [*BreakStatement* **break** [no line break] *Identifier*] **do**
    throw BREAKvalue: *d*, label: Name[*Identifier*]
end proc;

13.13 Return Statement

Syntax

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

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*$^{\text{allowIn}}$] **do**
    if *getRegionalFrame*(*env*) ≠ PARAMETERFRAME **then throw syntaxError end if**;
    Validate[*ListExpression*$^{\text{allowIn}}$](*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*$^{\text{allowIn}}$] **do**
    a: OBJECT ← *readReference*(Eval[*ListExpression*$^{\text{allowIn}}$](*env*, **run**), **run**);
    throw RETURNEDVALUE⟨value: *a*⟩
end proc;

13.14 Throw Statement

Syntax

ThrowStatement → **throw** [no line break] *ListExpression*$^{\text{allowIn}}$

Validation

proc Validate[*ThrowStatement* → **throw** [no line break] *ListExpression*$^{\text{allowIn}}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*ListExpression*$^{\text{allowIn}}$](*cxt*, *env*)
end proc;

Setup

proc Setup[*ThrowStatement* → **throw** [no line break] *ListExpression*$^{\text{allowIn}}$] ()
  Setup[*ListExpression*$^{\text{allowIn}}$]()
end proc;

Evaluation

proc Eval[*ThrowStatement* → **throw** [no line break] *ListExpression*$^{\text{allowIn}}$] (*env*: ENVIRONMENT): OBJECT
  a: OBJECT ← *readReference*(Eval[*ListExpression*$^{\text{allowIn}}$](*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*.

CompileEnv[*CatchClause*]: ENVIRONMENT;

CompileFrame[*CatchClause*]: LOCALFRAME;

proc Validate[*CatchClause* ⇒ **catch (** *Parameter* **)** *Block*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS)
compileFrame: LOCALFRAME ← **new** LOCALFRAME⟨⟨localBindings: {}, plurality: **plural**⟩⟩;
compileEnv: ENVIRONMENT ← **[***compileFrame***]** ⊕ *env*;
CompileFrame[*CatchClause*] ← *compileFrame*;
CompileEnv[*CatchClause*] ← *compileEnv*;
Validate[*Parameter*](*cxt*, *compileEnv*, *compileFrame*);
Validate[*Block*](*cxt*, *compileEnv*, *jt*, **plural**)
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*] ()
Setup[*Parameter*](CompileEnv[*CatchClause*], CompileFrame[*CatchClause*], **none**);
Setup[*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 Block1 CatchClausesOpt finally Block2] do
      result: OBJECT ∪ SEMANTICEXCEPTION;
      try result ← Eval[Block1](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[Block2](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);
   [CatchClauses0 ⇒ CatchClauses1 CatchClause] do
      r: OBJECT ∪ {reject} ← Eval[CatchClauses1](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**}
  compileFrame: LOCALFRAME ← CompileFrame[*CatchClause*];
  runtimeFrame: LOCALFRAME ← *instantiateLocalFrame*(*compileFrame*, *env*);
  runtimeEnv: ENVIRONMENT ← [*runtimeFrame*] ⊕ *env*;
  qname: QUALIFIEDNAME ← *public*::(Name[*Parameter*]);
  v: LOCALMEMBEROPT ← *findLocalMember*(*runtimeFrame*, {*qname*}, **write**);
  note Validate created one local variable with the name in *qname*, so *v* ∈ VARIABLE.
  type: CLASS ← *getVariableType*(*v*);
  if *type*.is(*exception*) **then**
    writeLocalMember(*v*, *exception*, **run**);
    return Eval[*Block*](*runtimeEnv*, **undefined**)
  else return reject
  end if
end proc;

14 Directives

Syntax

Directive$^{\omega}$ ⇒
  EmptyStatement
 | *Statement*$^{\omega}$
 | *AnnotatableDirective*$^{\omega}$
 | *Attributes* [no line break] *AnnotatableDirective*$^{\omega}$
 | *Attributes* [no line break] **{** *Directives* **}**
 | *PackageDefinition*
 | *Pragma Semicolon*$^{\omega}$

AnnotatableDirective$^{\omega}$ ⇒
  ExportDefinition Semicolon$^{\omega}$
 | *VariableDefinition*$^{\text{allowIn}}$ *Semicolon*$^{\omega}$
 | *FunctionDefinition*
 | *ClassDefinition*
 | *NamespaceDefinition Semicolon*$^{\omega}$
 | *ImportDirective Semicolon*$^{\omega}$
 | *UseDirective Semicolon*$^{\omega}$

Directives ⇒
  «empty»
 | *DirectivesPrefix Directive*$^{\text{abbrev}}$

DirectivesPrefix ⇒
  «empty»
 | *DirectivesPrefix Directive*$^{\text{full}}$

Validation

Enabled[*Directive*$^{\omega}$]: BOOLEAN;

proc Validate[*Directive*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS, *pl*: PLURALITY,
attr: ATTRIBUTEOPTNOTFALSE)
[*Directive* ⇒ *EmptyStatement*] **do nothing**;
[*Directive* ⇒ *Statement*] **do**
if *attr* ∉ {**none**, **true**} **then throw syntaxError end if**;
Validate[*Statement*](*cxt*, *env*, {}, *jt*, *pl*);
[*Directive* ⇒ *AnnotatableDirective*] **do**
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*);
if *attr3* = **false then** Enabled[*Directive*] ← **false**
else
Enabled[*Directive*] ← **true**;
Validate[*AnnotatableDirective*](*cxt*, *env*, *pl*, *attr3*)
end if;
[*Directive* ⇒ *Attributes* [no line break] { *Directives* }] **do**
Validate[*Attributes*](*cxt*, *env*);
Setup[*Attributes*]();
attr2: ATTRIBUTE ← Eval[*Attributes*](*env*, **compile**);
attr3: ATTRIBUTE ← *combineAttributes*(*attr*, *attr2*);
if *attr3* = **false then** Enabled[*Directive*] ← **false**
else
Enabled[*Directive*] ← **true**;
localCxt: CONTEXT ← **new** CONTEXT⟨⟨strict: *cxt*.strict, openNamespaces: *cxt*.openNamespaces,
constructsSuper: *cxt*.constructsSuper⟩⟩;
Validate[*Directives*](*localCxt*, *env*, *jt*, *pl*, *attr3*);
cxt.constructsSuper ← *localCxt*.constructsSuper
end if;
[*Directive* ⇒ *PackageDefinition*] **do**
if *attr* ∈ {**none**, **true**} **then** ???? **else throw syntaxError end if**;
[*Directive* ⇒ *Pragma Semicolon*] **do**
if *attr* ∈ {**none**, **true**} **then** Validate[*Pragma*](*cxt*) **else throw syntaxError end if**
end proc;

proc Validate[*AnnotatableDirective*]
(*cxt*: CONTEXT, *env*: ENVIRONMENT, *pl*: PLURALITY, *attr*: ATTRIBUTEOPTNOTFALSE)
[*AnnotatableDirective* ⇒ *ExportDefinition Semicolon*] **do** ????;
[*AnnotatableDirective* ⇒ *VariableDefinition*$^{\text{allowIn}}$ *Semicolon*] **do**
Validate[*VariableDefinition*$^{\text{allowIn}}$](*cxt*, *env*, *attr*);
[*AnnotatableDirective* ⇒ *FunctionDefinition*] **do**
Validate[*FunctionDefinition*](*cxt*, *env*, *pl*, *attr*);
[*AnnotatableDirective* ⇒ *ClassDefinition*] **do**
Validate[*ClassDefinition*](*cxt*, *env*, *pl*, *attr*);
[*AnnotatableDirective* ⇒ *NamespaceDefinition Semicolon*] **do**
Validate[*NamespaceDefinition*](*cxt*, *env*, *pl*, *attr*);

[*AnnotatableDirective*^ω ⇒ *ImportDirective Semicolon*^ω] **do** ????;
[*AnnotatableDirective*^ω ⇒ *UseDirective Semicolon*^ω] **do**
if *attr* ∈ {**none**, **true**} **then** Validate[*UseDirective*](*cxt*, *env*)
else throw syntaxError
end if
end proc;

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

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

Setup

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

proc Setup[*AnnotatableDirective*^ω] ()
[*AnnotatableDirective*^ω ⇒ *ExportDefinition Semicolon*^ω] **do** ????;
[*AnnotatableDirective*^ω ⇒ *VariableDefinition*^allowIn *Semicolon*^ω] **do**
Setup[*VariableDefinition*^allowIn]();
[*AnnotatableDirective*^ω ⇒ *FunctionDefinition*] **do** Setup[*FunctionDefinition*]();
[*AnnotatableDirective*^ω ⇒ *ClassDefinition*] **do** Setup[*ClassDefinition*]();
[*AnnotatableDirective*^ω ⇒ *NamespaceDefinition Semicolon*^ω] **do nothing**;
[*AnnotatableDirective*^ω ⇒ *ImportDirective Semicolon*^ω] **do** ????;
[*AnnotatableDirective*^ω ⇒ *UseDirective Semicolon*^ω] **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*$^{\text{allowIn}}$ *Semicolon*□] **do**
    return Eval[*VariableDefinition*$^{\text{allowIn}}$](*env*, *d*);
  [*AnnotatableDirective*□ □ *FunctionDefinition*] **do return** *d*;
  [*AnnotatableDirective*□ □ *ClassDefinition*] **do return** Eval[*ClassDefinition*](*env*, *d*);
  [*AnnotatableDirective*□ □ *NamespaceDefinition Semicolon*□] **do return** *d*;
  [*AnnotatableDirective*□ □ *ImportDirective Semicolon*□] **do** ????;
  [*AnnotatableDirective*□ □ *UseDirective Semicolon*□] **do return** *d*
end proc;

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

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

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 public;
   [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, enumerable: 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, enumerable: 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)
   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;
   cxt.openNamespaces ← cxt.openNamespaces ∪ namespaces
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)
  Validate[*PragmaItems*](*cxt*)
end proc;

Validate[*PragmaItems*] (*cxt*: CONTEXT) propagates the call to Validate to every nonterminal in the expansion of *PragmaItems*.

proc Validate[*PragmaItem*] (*cxt*: CONTEXT)
  [*PragmaItem* → *PragmaExpr*] **do** Validate[*PragmaExpr*](*cxt*, **false**);
  [*PragmaItem* → *PragmaExpr* **?**] **do** Validate[*PragmaExpr*](*cxt*, **true**)
end proc;

proc Validate[*PragmaExpr*] (*cxt*: CONTEXT, *optional*: BOOLEAN)
  [*PragmaExpr* → *Identifier*] **do**
    processPragma(*cxt*, Name[*Identifier*], **undefined**, *optional*);
  [*PragmaExpr* → *Identifier* **(** *PragmaArgument* **)**] **do**
    arg: OBJECT ← Value[*PragmaArgument*];
    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**] = $(-2^{63})_{\text{long}}$;
  Value[*PragmaArgument* → **String**] = Value[**String**];

proc *processPragma*(*cxt*: CONTEXT, *name*: STRING, *value*: OBJECT, *optional*: BOOLEAN)
  if *name* = “strict” **then**
    if *value* ∈ {**true**, **undefined**} **then** *cxt*.strict ← **true**; **return end if**;
    if *value* = **false then** *cxt*.strict ← **false**; **return end if**
  end if;
  if *name* = “ecmascript” **then**
    if *value* ∈ {**undefined**, 4.0_{f64}} **then return 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
    end if
  end if;
  if not *optional* **then 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*□

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*□]
  (*cxt*: CONTEXT, *env*: ENVIRONMENT, *attr*: ATTRIBUTEOPTNOTFALSE)
  Validate[*VariableBindingList*□](*cxt*, *env*, *attr*, Immutable[*VariableDefinitionKind*], **false**)
end proc;

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

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

CompileEnv[*VariableBinding*]: ENVIRONMENT;

CompileVar[*VariableBinding*]: VARIABLE DYNAMICVAR INSTANCEVARIABLE;

OverriddenVar[*VariableBinding*]: INSTANCEVARIABLEOPT;

Multiname[*VariableBinding*]: MULTINAME;

```
proc Validate[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□] (cxt: CONTEXT, env: ENVIRONMENT,
      attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN, noInitialiser: BOOLEAN)
   Validate[TypedIdentifier□](cxt, env);
   Validate[VariableInitialisation□](cxt, env);
   CompileEnv[VariableBinding□] □ env;
   name: STRING □ Name[TypedIdentifier□];
   if not cxt.strict and getRegionalFrame(env) □ PACKAGE □ PARAMETERFRAME and not immutable and
         attr = none and Plain[TypedIdentifier□] then
      qname: QUALIFIEDNAME □ public::name;
      Multiname[VariableBinding□] □ {qname};
      CompileVar[VariableBinding□] □ 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
            initialiser: INITIALISEROPT □ Initialiser[VariableInitialisation□];
            if noInitialiser and initialiser ≠ none then throw syntaxError end if;
            proc variableSetup(): CLASSOPT
               type: CLASSOPT □ SetupAndEval[TypedIdentifier□](env);
               Setup[VariableInitialisation□]();
               return type
            end proc;
            v: VARIABLE □ new VARIABLE□□type: uninitialised, value: uninitialised, immutable: immutable,
                  setup: variableSetup, initialiser: initialiser, initialiserEnv: env□□
            multiname: MULTINAME □ defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit,
                  readWrite, v);
            Multiname[VariableBinding□] □ multiname;
            CompileVar[VariableBinding□] □ v;
         {virtual, final} do
            note not noInitialiser;
            c: CLASS □ env[0];
            v: INSTANCEVARIABLE □ new INSTANCEVARIABLE□□final: memberMod = final,
                  enumerable: a.enumerable, immutable: immutable□□
            OverriddenVar[VariableBinding□] □ defineInstanceMember(c, cxt, name, a.namespaces, a.overrideMod,
                  a.explicit, v);
            CompileVar[VariableBinding□] □ v;
         {constructor} do throw definitionError
      end case
   end if
end proc;
```

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

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*□] ()
  Setup[*VariableBindingList*□]()
end proc;

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

proc Setup[*VariableBinding*□ □ *TypedIdentifier*□ *VariableInitialisation*□] ()
  env: ENVIRONMENT □ CompileEnv[*VariableBinding*□];
  v: VARIABLE □ DYNAMICVAR □ INSTANCEVARIABLE □ CompileVar[*VariableBinding*□];
  case *v* **of**
    VARIABLE **do**
      setupVariable(*v*);
      if not *v*.immutable **then** *v*.value □ *getVariableType*(*v*).defaultValue **end if**;
    DYNAMICVAR **do** Setup[*VariableInitialisation*□]();
    INSTANCEVARIABLE **do**
      t: CLASSOPT □ SetupAndEval[*TypedIdentifier*□](*env*);
      if *t* = **none then**
        overriddenVar: INSTANCEVARIABLEOPT □ OverriddenVar[*VariableBinding*□];
        if *overriddenVar* ≠ **none then** *t* □ *overriddenVar*.type
        else *t* □ *objectClass*
        end if
      end if;
      v.type □ *t*;
      Setup[*VariableInitialisation*□]();
      initialiser: INITIALISEROPT □ Initialiser[*VariableInitialisation*□];
      defaultValue: OBJECTU □ **undefined**;
      if *initialiser* ≠ **none then** *defaultValue* □ *initialiser*(*env*, **compile**)
      elsif not *v*.immutable **then** *defaultValue* □ *t*.defaultValue
      end if;
      v.defaultValue □ *defaultValue*
  end case
end proc;

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

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

Evaluation

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

Eval[VariableBindingList□] (env: ENVIRONMENT) propagates the call to Eval to every nonterminal in the expansion of VariableBindingList□.

```
proc Eval[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□] (env: ENVIRONMENT)
   case CompileVar[VariableBinding□] of
      VARIABLE do
         innerFrame: NONWITHFRAME □ env[0];
         members: LOCALMEMBER{} □ {b.content | □b □ innerFrame.localBindings such that
               b.qname □ Multiname[VariableBinding□]};
         note The members set consists of exactly one VARIABLE element because innerFrame was constructed with that
               VARIABLE inside Validate.
         v: VARIABLE □ the one element of members;
         initialiser: INITIALISER □ {none, busy} □ v.initialiser;
         case initialiser of
            {none} do nothing;
            {busy} do throw propertyAccessError;
            INITIALISER do
               v.initialiser □ busy;
               value: OBJECT □ initialiser(v.initialiserEnv, run);
               writeVariable(v, value, true)
         end case;
      DYNAMICVAR do
         initialiser: INITIALISEROPT □ Initialiser[VariableInitialisation□];
         if initialiser ≠ none then
            value: OBJECT □ initialiser(env, run);
            lexicalWrite(env, Multiname[VariableBinding□], value, false, run)
         end if;
      INSTANCEVARIABLE do nothing
   end case
end proc;
```

```
proc WriteBinding[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□]
      (env: ENVIRONMENT, newValue: OBJECT)
   case CompileVar[VariableBinding□] of
      VARIABLE do
         innerFrame: NONWITHFRAME □ env[0];
         members: LOCALMEMBER{} □ {b.content | □b □ innerFrame.localBindings such that
               b.qname □ Multiname[VariableBinding□]};
         note The members set consists of exactly one VARIABLE element because innerFrame was constructed with that
               VARIABLE inside Validate.
         v: VARIABLE □ the one element of members;
         writeVariable(v, newValue, false);
      DYNAMICVAR do
         lexicalWrite(env, Multiname[VariableBinding□], newValue, false, run)
   end case
end proc;
```

Initialiser[*VariableInitialisation*]: INITIALISEROPT;
  Initialiser[*VariableInitialisation* «empty»] = **none**;
  Initialiser[*VariableInitialisation* **=** *VariableInitialiser*] = Eval[*VariableInitialiser*];

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* instead of a *Directive* in non-strict mode. In strict mode variable definitions may not be used as substatements.

SimpleVariableDefinition **var** *UntypedVariableBindingList*

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

UntypedVariableBinding *Identifier* *VariableInitialisation*$^{\mathrm{allowIn}}$

Validation

proc Validate[*SimpleVariableDefinition* **var** *UntypedVariableBindingList*] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  if *cxt*.strict **or** *getRegionalFrame*(*env*) PACKAGE 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*$^{\mathrm{allowIn}}$] (*cxt*: CONTEXT, *env*: ENVIRONMENT)
  Validate[*VariableInitialisation*$^{\mathrm{allowIn}}$](*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*$^{\text{allowIn}}$] ()
  Setup[*VariableInitialisation*$^{\text{allowIn}}$]()
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*$^{\text{allowIn}}$] (*env*: ENVIRONMENT)
  initialiser: INITIALISEROPT ← Initialiser[*VariableInitialisation*$^{\text{allowIn}}$];
  if *initialiser* ≠ **none then**
    value: OBJECT ← *initialiser*(*env*, **run**);
    qname: QUALIFIEDNAME ← *public*::(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*]: NONWITHFRAME;

OverriddenMethod[*FunctionDefinition*]: INSTANCEMETHODOPT;

```
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 then throw definitionError end if;
   localCxt: CONTEXT ← new CONTEXT⟨⟨strict: cxt.strict, openNamespaces: cxt.openNamespaces,
         constructsSuper: none⟩⟩;
   case memberMod of
      {none, static} do
         f: SIMPLEINSTANCE ∪ UNINSTANTIATEDFUNCTION;
         if kind ∈ {get, set} then ????
         else
            this: {none, uninitialised} ← prototype ? uninitialised : none;
            f ← ValidateStaticFunction[FunctionCommon](localCxt, env, this, unchecked, prototype)
         end if;
         if pl = singular then f ← instantiateFunction(f, env) end if;
         if unchecked and attr = none and
               (env[0] ∈ PACKAGE or (env[0] ∈ LOCALFRAME and env[1] ∈ PARAMETERFRAME)) then
            defineHoistedVar(env, name, f)
         else
            v: VARIABLE ← new VARIABLE⟨⟨type: functionClass, value: f, immutable: true, setup: none,
                  initialiser: none, initialiserEnv: env⟩⟩;
            defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
         end if;
         OverriddenMethod[FunctionDefinition] ← none;
      {virtual, final} do
         note pl = singular;
         if prototype then throw definitionError end if;
         if kind ∈ {get, set} then ???? end if;
         Validate[FunctionCommon](localCxt, env, uninitialised, false, prototype);
         method: INSTANCEMETHOD ← new INSTANCEMETHOD⟨⟨final: memberMod = final,
               enumerable: a.enumerable, signature: CompileFrame[FunctionCommon],
               call: EvalInstanceCall[FunctionCommon], env: env⟩⟩;
         OverriddenMethod[FunctionDefinition] ← defineInstanceMember(env[0], cxt, name, a.namespaces,
               a.overrideMod, a.explicit, method);
      {constructor} do
         note pl = singular;
         if prototype then throw definitionError end if;
         OverriddenMethod[FunctionDefinition] ← none;
         ????
   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**, **uninitialised**}, *unchecked*: Boolean, *prototype*: Boolean)
  compileFrame: ParameterFrame ← **new** ParameterFrame⟨⟨localBindings: {}, plurality: **plural**, this: *this*,
      unchecked: *unchecked*, prototype: *prototype*, parameters: **[]**, rest: **none**⟩⟩;
  compileEnv: Environment ← **[***compileFrame***]** ⊕ *env*;
  CompileFrame[*FunctionCommon*] ← *compileFrame*;
  CompileEnv[*FunctionCommon*] ← *compileEnv*;
  if *unchecked* **then** *defineHoistedVar*(*compileEnv*, "`arguments`", **undefined**) **end if**;
  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**, **uninitialised**}, *unchecked*: Boolean, *prototype*: Boolean): UninstantiatedFunction
  Validate[*FunctionCommon*](*cxt*, *env*, *this*, *unchecked*, *prototype*);
  length: Integer ← ParameterCount[*Parameters*];
  if *prototype* **then**
    return new UninstantiatedFunction⟨⟨type: *prototypeFunctionClass*, buildPrototype: **true**, length: *length*,
        call: EvalStaticCall[*FunctionCommon*], construct: EvalPrototypeConstruct[*FunctionCommon*],
        instantiations: {}⟩⟩
  else
    return new UninstantiatedFunction⟨⟨type: *functionClass*, buildPrototype: **false**, length: *length*,
        call: EvalStaticCall[*FunctionCommon*], construct: **none**, instantiations: {}⟩⟩
  end if
end proc;

Setup

proc Setup[*FunctionDefinition* ⇒ `function` *FunctionName FunctionCommon*] ()
  overriddenMethod: InstanceMethodOpt ← OverriddenMethod[*FunctionDefinition*];
  if *overriddenMethod* ≠ **none then**
    SetupOverride[*FunctionCommon*](*overriddenMethod*.signature)
  else Setup[*FunctionCommon*]()
  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, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
   if phase = compile then throw compileExpressionError end if;
   runtimeEnv: ENVIRONMENT □ f.env;
   runtimeThis: OBJECTOPT □ none;
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   if compileFrame.prototype then
      if this □ PRIMITIVEOBJECT then runtimeThis □ getPackageFrame(runtimeEnv)
      else runtimeThis □ this
      end if
   end if;
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
   assignArguments(runtimeFrame, f, args, phase);
   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 EvalInstanceCall[FunctionCommon ⇒ ( Parameters ) Result Block]
      (this: OBJECT, args: OBJECT[], env: ENVIRONMENT, phase: PHASE): OBJECT
   if phase = compile then throw compileExpressionError end if;
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, env, this);
   note not runtimeFrame.unchecked;
   assignArguments(runtimeFrame, none, args, phase);
   result: OBJECT;
   try Eval[Block]([runtimeFrame] ⊕ env, 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]
      (f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
   if phase = compile then throw compileExpressionError end if;
   runtimeEnv: ENVIRONMENT ← f.env;
   super: OBJECT ← dotRead(f, {public::"prototype"}, phase);
   if super ∈ {null, undefined} then super ← objectPrototype
   elsif not prototypeClass.is(super) then throw badValueError
   end if;
   o: OBJECT ← createSimpleInstance(prototypeClass, super, none, none, none);
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, runtimeEnv, o);
   assignArguments(runtimeFrame, f, args, phase);
   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);
   if coercedResult ∈ PRIMITIVEOBJECT then return o else return coercedResult end if
end proc;
```

proc *assignArguments*(*runtimeFrame*: PARAMETERFRAME, *f*: SIMPLEINSTANCE {**none**}, *args*: OBJECT[], *phase*: {**run**})

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.

```
argumentsObject: OBJECTOPT  none;
if runtimeFrame.unchecked then
    argumentsObject  arrayClass.construct([], phase);
    createDynamicProperty(argumentsObject, public::“callee”, false, false, f);
    nArgs: INTEGER  |args|;
    if nArgs > arrayLimit then throw rangeError end if;
    dotWrite(argumentsObject, {arrayPrivate::“length”}, nArgs_ulong, phase)
end if;
restObject: OBJECTOPT  none;
rest: VARIABLE  {none}  runtimeFrame.rest;
if rest ≠ none then restObject  arrayClass.construct([], phase) end if;
parameters: PARAMETER[]  runtimeFrame.parameters;
i: INTEGER  0;
j: INTEGER  0;
for each arg  args do
    if i < |parameters| then
        parameter: PARAMETER  parameters[i];
        v: DYNAMICVAR  VARIABLE  parameter.var;
        writeLocalMember(v, arg, phase);
        if argumentsObject ≠ none then
            note Create an alias of v as the ith entry of the arguments object.
            note v  DYNAMICVAR;
            qname: QUALIFIEDNAME  toQualifiedName(i_ulong, phase);
            argumentsObject.localBindings  argumentsObject.localBindings  {LOCALBINDINGqname: qname,
                    accesses: readWrite, content: v, explicit: false, enumerable: false}
        end if
    elsif restObject ≠ none then
        if j ≥ arrayLimit then throw rangeError end if;
        indexWrite(restObject, j, arg, phase);
        note argumentsObject = none because a function can't have both a rest parameter and an arguments object.
        j  j + 1
    elsif argumentsObject ≠ none then indexWrite(argumentsObject, i, arg, phase)
    else throw argumentMismatchError
    end if;
    i  i + 1
end for each;
while i < |parameters| do
    parameter: PARAMETER  parameters[i];
    default: OBJECTOPT  parameter.default;
    if default = none then
        if argumentsObject ≠ none then default  undefined
        else throw argumentMismatchError
        end if
    end if;
    writeLocalMember(parameter.var, default, phase);
    i  i + 1
end while
end proc;
```

Syntax

Parameters □
  «empty»
| *NonemptyParameters*

NonemptyParameters □
  ParameterInit
| *ParameterInit* **,** *NonemptyParameters*
| *RestParameter*

Parameter □ *ParameterAttributes TypedIdentifier*$^{\text{allowIn}}$

ParameterAttributes □
  «empty»
| **const**

ParameterInit □
  Parameter
| *Parameter* **=** *AssignmentExpression*$^{\text{allowIn}}$

RestParameter □
  ...
| **...** *ParameterAttributes Identifier*

Result □
  «empty»
| **:** *TypeExpression*$^{\text{allowIn}}$

Validation

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

ParameterCount[*Parameters*]: INTEGER;
  ParameterCount[*Parameters* □ «empty»] = 0;
  ParameterCount[*Parameters* □ *NonemptyParameters*] = ParameterCount[*NonemptyParameters*];

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

Plain[*NonemptyParameters*]: BOOLEAN;
  Plain[*NonemptyParameters* □ *ParameterInit*] = Plain[*ParameterInit*];
  Plain[*NonemptyParameters*$_0$ □ *ParameterInit* **,** *NonemptyParameters*$_1$]
    = Plain[*ParameterInit*] **and** Plain[*NonemptyParameters*$_1$];
  Plain[*NonemptyParameters* □ *RestParameter*] = **false**;

ParameterCount[*NonemptyParameters*]: INTEGER;
  ParameterCount[*NonemptyParameters* □ *ParameterInit*] = 1;
  ParameterCount[*NonemptyParameters*$_0$ □ *ParameterInit* **,** *NonemptyParameters*$_1$]
    = 1 + ParameterCount[*NonemptyParameters*$_1$];
  ParameterCount[*NonemptyParameters* □ *RestParameter*] = 0;

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

Name[*Parameter* → *ParameterAttributes TypedIdentifier*^{allowIn}]: STRING = Name[*TypedIdentifier*^{allowIn}];

Plain[*Parameter* → *ParameterAttributes TypedIdentifier*^{allowIn}]: BOOLEAN
    = Plain[*TypedIdentifier*^{allowIn}] **and not** HasConst[*ParameterAttributes*];

CompileVar[*Parameter*]: DYNAMICVAR ∪ VARIABLE;

proc Validate[*Parameter* → *ParameterAttributes TypedIdentifier*^{allowIn}]
    (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME ∪ LOCALFRAME)
  Validate[*TypedIdentifier*^{allowIn}](*cxt*, *env*);
  immutable: BOOLEAN ← HasConst[*ParameterAttributes*];
  name: STRING ← Name[*TypedIdentifier*^{allowIn}];
  v: DYNAMICVAR ∪ VARIABLE;
  if *compileFrame* ∈ PARAMETERFRAME **and** *compileFrame*.unchecked **then**
    note not *immutable*;
    v ← *defineHoistedVar*(*env*, *name*, **undefined**)
  else
    v ← **new** VARIABLE⟨⟨type: **uninitialised**, value: **uninitialised**, immutable: *immutable*, setup: **none**,
        initialiser: **none**, initialiserEnv: *env*⟩⟩;
    defineLocalMember(*env*, *name*, {*public*}, **none**, **false**, **readWrite**, *v*)
  end if;
  CompileVar[*Parameter*] ← *v*
end proc;

HasConst[*ParameterAttributes*]: BOOLEAN;
  HasConst[*ParameterAttributes* → «empty»] = **false**;
  HasConst[*ParameterAttributes* → **const**] = **true**;

Plain[*ParameterInit*]: BOOLEAN;
  Plain[*ParameterInit* → *Parameter*] = Plain[*Parameter*];
  Plain[*ParameterInit* → *Parameter* **=** *AssignmentExpression*^{allowIn}] = **false**;

proc Validate[*ParameterInit*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME)
  [*ParameterInit* → *Parameter*] **do** Validate[*Parameter*](*cxt*, *env*, *compileFrame*);
  [*ParameterInit* → *Parameter* **=** *AssignmentExpression*^{allowIn}] **do**
    Validate[*Parameter*](*cxt*, *env*, *compileFrame*);
    Validate[*AssignmentExpression*^{allowIn}](*cxt*, *env*)
end proc;

proc Validate[*RestParameter*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME)
  [*RestParameter* → **...**] **do**
    note not *compileFrame*.unchecked;
    v: VARIABLE ← **new** VARIABLE⟨⟨type: *arrayClass*, value: **uninitialised**, immutable: **true**, setup: **none**,
        initialiser: **none**, initialiserEnv: *env*⟩⟩;
    compileFrame.rest ← *v*;
  [*RestParameter* → **...** *ParameterAttributes Identifier*] **do**
    note not *compileFrame*.unchecked;
    v: VARIABLE ← **new** VARIABLE⟨⟨type: *arrayClass*, value: **uninitialised**,
        immutable: HasConst[*ParameterAttributes*], setup: **none**, initialiser: **none**, initialiserEnv: *env*⟩⟩;
    compileFrame.rest ← *v*;
    name: STRING ← Name[*Identifier*];
    defineLocalMember(*env*, *name*, {*public*}, **none**, **false**, **readWrite**, *v*)
end proc;

Plain[*Result*]: BOOLEAN;
  Plain[*Result* ⇒ «empty»] = **true**;
  Plain[*Result* ⇒ **:** *TypeExpression*$^{\text{allowIn}}$] = **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.parameters ≠ [] or overriddenSignature.rest ≠ none then
         throw definitionError
      end if;
   [Parameters ⇒ NonemptyParameters] do
      SetupOverride[NonemptyParameters](compileEnv, compileFrame, overriddenSignature,
            overriddenSignature.parameters)
end proc;

proc Setup[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
   [NonemptyParameters ⇒ ParameterInit] do
      Setup[ParameterInit](compileEnv, compileFrame);
   [NonemptyParameters0 ⇒ ParameterInit , NonemptyParameters1] do
      Setup[ParameterInit](compileEnv, compileFrame);
      Setup[NonemptyParameters1](compileEnv, compileFrame);
   [NonemptyParameters ⇒ RestParameter] do nothing
end proc;

proc SetupOverride[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
      overriddenSignature: PARAMETERFRAME, overriddenParameters: PARAMETER[])
   [NonemptyParameters ⇒ ParameterInit] do
      if overriddenParameters = [] then throw definitionError end if;
      SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]);
      if |overriddenParameters| ≠ 1 or overriddenSignature.rest ≠ none then
         throw definitionError
      end if;
   [NonemptyParameters0 ⇒ ParameterInit , NonemptyParameters1] do
      if overriddenParameters = [] then throw definitionError end if;
      SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]);
      SetupOverride[NonemptyParameters1](compileEnv, compileFrame, overriddenSignature,
            overriddenParameters[1 ...]);
   [NonemptyParameters ⇒ RestParameter] do
      if overriddenParameters ≠ [] then throw definitionError end if;
      overriddenRest: VARIABLE ∪ {none} ← overriddenSignature.rest;
      if overriddenRest = none or getVariableType(overriddenRest) ≠ arrayClass then
         throw definitionError
      end if
end proc;
```

```
proc Setup[Parameter □ ParameterAttributes TypedIdentifier^allowIn]
      (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME □ LOCALFRAME, default: OBJECTOPT)
   if compileFrame □ PARAMETERFRAME and default = none and
         (some p2 □ compileFrame.parameters satisfies p2.default ≠ none) then
      note A required parameter cannot follow an optional one.
      throw definitionError
   end if;
   v: DYNAMICVAR □ VARIABLE □ CompileVar[Parameter];
   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
   end case;
   if compileFrame □ PARAMETERFRAME then
      p: PARAMETER □ PARAMETER□var: v, default: default□;
      compileFrame.parameters □ compileFrame.parameters ⊕ [p]
   end if
end proc;

proc SetupOverride[Parameter □ ParameterAttributes TypedIdentifier^allowIn] (compileEnv: ENVIRONMENT,
      compileFrame: PARAMETERFRAME, default: OBJECTOPT, overriddenParameter: PARAMETER)
   newDefault: OBJECTOPT □ default;
   if newDefault = none then newDefault □ overriddenParameter.default end if;
   if default = none and (some p2 □ compileFrame.parameters satisfies p2.default ≠ none) then
      note A required parameter cannot follow an optional one.
      throw definitionError
   end if;
   v: DYNAMICVAR □ VARIABLE □ CompileVar[Parameter];
   note v □ DYNAMICVAR;
   type: CLASSOPT □ SetupAndEval[TypedIdentifier^allowIn](compileEnv);
   if type = none then type □ objectClass end if;
   if type ≠ getVariableType(overriddenParameter.var) then throw definitionError end if;
   v.type □ type;
   p: PARAMETER □ PARAMETER□var: v, default: newDefault□;
   compileFrame.parameters □ compileFrame.parameters ⊕ [p]
end proc;

proc Setup[ParameterInit] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
   [ParameterInit □ Parameter] do Setup[Parameter](compileEnv, compileFrame, none);
   [ParameterInit □ Parameter = AssignmentExpression^allowIn] do
      Setup[AssignmentExpression^allowIn]();
      default: OBJECT □ readReference(Eval[AssignmentExpression^allowIn](compileEnv, compile), compile);
      Setup[Parameter](compileEnv, compileFrame, default)
end proc;

proc SetupOverride[ParameterInit]
      (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenParameter: PARAMETER)
   [ParameterInit □ Parameter] do
      SetupOverride[Parameter](compileEnv, compileFrame, none, overriddenParameter);
```

[*ParameterInit* ⇒ *Parameter* **=** *AssignmentExpression*^{allowIn}] **do**
Setup[*AssignmentExpression*^{allowIn}]();
default: OBJECT ← *readReference*(Eval[*AssignmentExpression*^{allowIn}](*compileEnv*, **compile**), **compile**);
SetupOverride[*Parameter*](*compileEnv*, *compileFrame*, *default*, *overriddenParameter*)
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;
  super: Class ← Validate[Inheritance](cxt, env);
  if not super.complete or super.final then throw definitionError end if;
  a: CompoundAttribute ← toCompoundAttribute(attr);
  if a.prototype then throw definitionError 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 super.dynamic;
  c: Class ← new Class⟨⟨localBindings: {}, super: super, instanceMembers: {}, complete: false,
      prototype: super.prototype, typeofString: "object", privateNamespace: privateNamespace,
      dynamic: dynamic, final: final, defaultValue: null, bracketRead: super.bracketRead,
      bracketWrite: super.bracketWrite, bracketDelete: super.bracketDelete, read: super.read, write: super.write,
      delete: super.delete, enumerate: super.enumerate⟩⟩;
  proc cIs(o: Object): Boolean
    return isAncestor(c, objectType(o))
  end proc;
  c.is ← cIs;
  proc cImplicitCoerce(o: Object, silent: Boolean): Object
    if o = null or c.is(o) then return o
    elsif silent then return null
    else throw badValueError
    end if
  end proc;
  c.implicitCoerce ← cImplicitCoerce;
  proc cCall(this: Object, args: Object[], phase: Phase): Object
    if |args| ≠ 1 then throw argumentMismatchError end if;
    return cImplicitCoerce(args[0], false)
  end proc;
  c.call ← cCall;
  proc cConstruct(args: Object[], phase: Phase): Object
    constructor: Object ← dotRead(c, {public::(Name[Identifier])}, phase);
    return call(null, constructor, args, phase)
  end proc;
  c.construct ← cConstruct;
  Class[ClassDefinition] ← c;
  v: Variable ← new Variable⟨⟨type: classClass, value: c, immutable: true, setup: none, initialiser: none,
      initialiserEnv: env⟩⟩;
  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*$^{\text{allowIn}}$] **do**
    Validate[*TypeExpression*$^{\text{allowIn}}$](*cxt*, *env*);
    return SetupAndEval[*TypeExpression*$^{\text{allowIn}}$](*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**, setup: **none**, initialiser: **none**,
      initialiserEnv: *env*⟩⟩;
  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
    cxt: CONTEXT ← **new** CONTEXT⟨⟨strict: **false**, openNamespaces: {*public*}, constructsSuper: **none**⟩⟩;
    Validate[*Directives*](*cxt*, *initialEnvironment*, JUMPTARGETS⟨⟨breakTargets: {}, continueTargets: {}⟩⟩, **singular**, **none**);
    Setup[*Directives*]();
    return Eval[*Directives*](*initialEnvironment*, **undefined**)
  end;

17 Predefined Identifiers

18 Built-in Classes

```
proc makeBuiltInClass(super: ClassOpt, prototype: ObjectOpt, typeofString: String, dynamic: Boolean,
      allowNull: Boolean, final: Boolean, defaultValue: Object,
      bracketRead: Object  Class  Object[]  Phase   ObjectOpt,
      bracketWrite: Object  Class  Object[]  Object  {run}   {none, ok},
      bracketDelete: Object  Class  Object[]  {run}   BooleanOpt,
      read: Object  Class  Multiname  LookupKind  Phase   ObjectOpt,
      write: Object  Class  Multiname  LookupKind  Boolean  Object  {run}   {none, ok},
      delete: Object  Class  Multiname  LookupKind  {run}   BooleanOpt,
      enumerate: Object   Object{}): Class
   proc call(this: Object, args: Object[], phase: Phase): Object
      ????
   end proc;
   proc construct(args: Object[], phase: Phase): Object
      ????
   end proc;
   privateNamespace: Namespace   new Namespace name: “private” ;
   c: Class   new Class localBindings: {}, super: super, instanceMembers: {}, complete: true,
         prototype: prototype, typeofString: typeofString, privateNamespace: privateNamespace, dynamic: dynamic,
         final: final, defaultValue: defaultValue, bracketRead: bracketRead, bracketWrite: bracketWrite,
         bracketDelete: bracketDelete, read: read, write: write, delete: delete, enumerate: enumerate, call: call,
         construct: construct ;
   proc is(o: Object): Boolean
      return isAncestor(c, objectType(o))
   end proc;
   c.is   is;
   proc implicitCoerce(o: Object, silent: Boolean): Object
      if c.is(o) or (o = null and allowNull) then return o
      elsif silent and allowNull then return null
      else throw badValueError
      end if
   end proc;
   c.implicitCoerce   implicitCoerce;
   return c
end proc;

proc makeSimpleBuiltInClass(super: Class, typeofString: String, dynamic: Boolean, allowNull: Boolean,
      final: Boolean, defaultValue: Object): Class
   return makeBuiltInClass(super, super.prototype, typeofString, dynamic, allowNull, final, defaultValue,
         super.bracketRead, super.bracketWrite, super.bracketDelete, super.read, super.write, super.delete,
         super.enumerate)
end proc;
```

proc *makeBuiltInIntegerClass*(*low*: INTEGER, *high*: INTEGER): CLASS
  proc *call*(*this*: OBJECT, *args*: OBJECT[], *phase*: PHASE): OBJECT
    ????
  end proc;
  proc *construct*(*args*: OBJECT[], *phase*: PHASE): OBJECT
    ????
  end proc;
  proc *is*(*o*: OBJECT): BOOLEAN
    if *o* ∈ FLOAT64 **then**
      case *o* **of**
        {**NaN**$_{f64}$, **+∞**$_{f64}$, **–∞**$_{f64}$} **do return false**;
        {**+zero**$_{f64}$, **–zero**$_{f64}$} **do return true**;
        NONZEROFINITEFLOAT64 **do**
          r: RATIONAL ← *o*.value;
          return *r* ∈ INTEGER **and** *low* ≤ *r* ≤ *high*
      end case
    else return false
    end if
  end proc;
  proc *implicitCoerce*(*o*: OBJECT, *silent*: BOOLEAN): OBJECT
    if *o* = **undefined then return +zero**$_{f64}$
    elsif *o* ∈ GENERALNUMBER **then**
      i: INTEGEROPT ← *checkInteger*(*o*);
      if *i* ≠ **none and** *low* ≤ *i* ≤ *high* **then**
        note **–zero**$_{f32}$, **+zero**$_{f32}$, and **–zero**$_{f64}$ are all coerced to **+zero**$_{f64}$.
        return *realToFloat64*(*i*)
      end if
    end if;
    throw badValueError
  end proc;
  privateNamespace: NAMESPACE ← **new** NAMESPACE⟨⟨name: "`private`"⟩⟩;
  return new CLASS⟨⟨localBindings: {}, super: *numberClass*, instanceMembers: {}, complete: **true**,
      prototype: *numberClass*.prototype, typeofString: "`number`", privateNamespace: *privateNamespace*,
      dynamic: **false**, final: **true**, defaultValue: **+zero**$_{f64}$, bracketRead: *numberClass*.bracketRead,
      bracketWrite: *numberClass*.bracketWrite, bracketDelete: *numberClass*.bracketDelete,
      read: *numberClass*.read, write: *numberClass*.write, delete: *numberClass*.delete,
      enumerate: *numberClass*.enumerate, call: *call*, construct: *construct*, is: *is*, implicitCoerce: *implicitCoerce*⟩⟩
end proc;

objectClass: CLASS = *makeBuiltInClass*(**none**, **none**, "`object`", **false**, **true**, **false**, **undefined**, *defaultBracketRead*, *defaultBracketWrite*, *defaultBracketDelete*, *defaultReadProperty*, *defaultWriteProperty*, *defaultDeleteProperty*, *defaultEnumerate*);

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

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

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

generalNumberClass: CLASS = *makeSimpleBuiltInClass*(*objectClass*, "`object`", **false**, **false**, **false**, **NaN**$_{f64}$);

longClass: CLASS = *makeSimpleBuiltInClass*(*generalNumberClass*, "`long`", **false**, **false**, **true**, 0_{long});

uLongClass: CLASS = *makeSimpleBuiltInClass*(*generalNumberClass*, "`ulong`", **false**, **false**, **true**, 0_{ulong});

floatClass: CLASS = *makeSimpleBuiltInClass*(*generalNumberClass*, "`float`", **false**, **false**, **true**, **NaN**$_{f32}$);

numberClass: CLASS = *makeSimpleBuiltInClass*(*generalNumberClass*, "`number`", **false**, **false**, **true**, **NaN**$_{f64}$);

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 = *makeSimpleBuiltInClass*(*objectClass*, “`character`”, **false**, **false**, **true**, ‘«NUL»’);

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

arrayClass: CLASS = *makeBuiltInClass*(*objectClass*, *arrayPrototype*, “`object`”, **true**, **true**, **true**, **null**,
defaultBracketRead, *defaultBracketWrite*, *defaultBracketDelete*, *defaultReadProperty*, *arrayWriteProperty*,
defaultDeleteProperty, *defaultEnumerate*);

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

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

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

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

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

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

prototypeFunctionClass: CLASS = *makeSimpleBuiltInClass*(*functionClass*, “`function`”, **true**, **true**, **true**, **null**);

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

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

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

arrayPrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE⟨⟨localBindings: {}, super: *objectPrototype*, sealed: **false**,
type: *arrayClass*, slots: {}, call: **none**, construct: **none**, env: **none**⟩⟩;

arrayLimit: INTEGER = $2^{64} - 1$;

arrayPrivate: NAMESPACE = **new** NAMESPACE⟨⟨name: “`private`”⟩⟩;

proc *arrayWriteProperty*(*o*: OBJECT, *limit*: CLASS, *multiname*: MULTINAME, *kind*: LOOKUPKIND,
createIfMissing: BOOLEAN, *newValue*: OBJECT, *phase*: {**run**}): {**none**, **ok**}
result: {**none**, **ok**} □ *defaultWriteProperty*(*o*, *limit*, *multiname*, *kind*, *createIfMissing*, *newValue*, *phase*);
if *result* = **ok** **and** |*multiname*| = 1 **then**
qname: QUALIFIEDNAME □ the one element of *multiname*;
if *qname*.namespace = *public* **then**
name: STRING □ *qname*.id;
i: INTEGER □ *truncateToInteger*(*toGeneralNumber*(*name*, *phase*));
if *name* = *integerToString*(*i*) **and** 0 ≤ *i* < *arrayLimit* **then**
length: ULONG □ *readInstanceProperty*(*o*, *arrayPrivate*::“`length`”, *phase*);
if *i* ≥ *length*.value **then**
length □ $(i + 1)_{\textbf{ulong}}$;
dotWrite(*o*, {*arrayPrivate*::“`length`”}, *length*, *phase*)
end if
end if
end if
end if;
return *result*
end proc;

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