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. *****Fix me

When denoted in this specification, characters with ~~values~~ code points between 20 and 7E hexadecimal inclusive are in a `fixed width font`. Unicode characters in the Basic Multilingual Plane (code points from 0 to FFFF hexadecimal)~~Other characters~~ are denoted by enclosing their four-digit hexadecimal Unicode code points~~value~~ between «u and ». Supplementary Unicode characters (code points from 10000 to 10FFFF hexadecimal) are denoted by enclosing their eight-digit hexadecimal Unicode code points between «U and ». For example, the non-breakable space character would be denoted in this document as «u00A0», and the character with the code point 1234F hexadecimal would be denoted as «U0001234F». 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 The **and** and **or** operators short-circuit. These are the only operators that do not always compute all of their operands.

5.5 Sets

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

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

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

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

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

Sets of either integers or characters can be abbreviated using the ... range operator, which generates inclusive ranges of integers or character code points. For example, the above set can also be written as $\{0, –5, 3 \ldots 3, 10 \ldots 13\}$.

If the beginning of the range is equal to the end of the range, then the range consists of only one element: $\{7 \ldots 7\}$ is the same as $\{7\}$. If the end of the range is one less than the beginning, then the range contains no elements: $\{7 \ldots 6\}$ 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,

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

The same notation is used for operations on sets and on semantic domains. Let A and B be sets (or semantic domains) and x and y be values. The following operations can be done on them:

$x \in A$	**true** if x is an element of A and **false** if not
$x \notin A$	**false** if x is an element of A and **true** if not
$\lvert A \rvert$	The number of elements in A (only used on finite sets)
min A	The value m that satisfies both $m \in A$ and for all elements $x \in A$, $x \geq m$ (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 $\mathrm{T}\{\}$ is the semantic domain of all sets whose elements are members of T. For example, if

$\mathrm{T} = \{1,2,3\}$

then:

$\mathrm{T}\{\} = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

The empty set $\{\}$ is a member of $\mathrm{T}\{\}$ 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 yth power (used only when either $x \neq 0$ and y is an integer or x is any number and $y>0$)
$\lvert x\rvert$	The absolute value of x, which is x if $x \geq 0$ and $-x$ otherwise
$\lfloor x \rfloor$	*Floor* of x, which is the unique integer i such that $i \leq x < i+1$. $\lfloor \pi \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, and $\lfloor 7 \rfloor = 7$.
$\lceil x \rceil$	*Ceiling* of x, which is the unique integer i such that $i-1 < x \leq i$. $\lceil \pi \rceil = 4$, $\lceil -3.5 \rceil = -3$, and $\lceil 7 \rceil = 7$.
x **mod** y	x modulo y, which is defined as $x - y \times \lfloor x/y \rfloor$. y must not be zero. 10 **mod** 7 = 3, and –1 **mod** 7 = 6.
$\log_{10}(x)$	The exact base-10 logarithm of x (x will always be greater than zero)

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

5.6.1 Bitwise Integer Operators

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

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

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

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

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

If x is zero or positive, then its sequence will have infinitely many consecutive leading 0's, while a negative integer x will generate a sequence with infinitely many consecutive leading 1's. For example, 6 generates the sequence ...0...0000110, while –6 generates ...1...1111010.

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

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

5.7 Characters

Characters enclosed in single quotes ‘ and ’ represent single Unicode characters with code points ranging from 0000 to 10FFFF hexadecimal. Even though Unicode does not define characters for some of these code points, in this specification any of these 1114112 code points is considered to be a valid character~~16-bit code points~~. Examples of characters include ‘A’, ‘b’, ‘«LF»’, ~~and~~ ‘«uFFFF»’, ‘«U00010000»’ and , ‘«U0010FFFF»’ (see also section 5.1).~~ Unicode surrogates are considered to be pairs of characters for the purpose of this specification.~~

Unicode has the notion of *code points*, which are numerical indices of characters in the Unicode character table, as well as *code units*, which are numerical values for storing characters in a particular representation. JavaScript is designed to make it appear that strings are represented in the UTF-16 representation, which means that a code unit is a 16-bit value (an implementation may store strings in other formats such as UTF-8, but it must make it appear for indexing and character extraction purposes as if strings were sequences of 16-bit code units). For convenience this specification does not distinguish between code units and code points in the range from 0000 to FFFF hexadecimal.

~~CHARACTER~~ CHAR16 is the semantic domain of the 65536 Unicode characters in the set~~all 65536 characters~~ {‘«u0000»’ ... ‘«uFFFF»’}. These characters form Unicode's Basic Multilingual Plane. These characters have code points between 0000 and FFFF hexadecimal. Code units are also represented by values in the CHAR16 semantic domain.

SUPPLEMENTARYCHAR is the semantic domain of the 1048576 Unicode characters in the set {‘«U00010000»’ ... ‘«U0010FFFF»’}. These are Unicode's supplementary characters with code points between 10000 and 10FFFF hexadecimal. Since these characters are not members of the CHAR16 domain, they cannot be stored directly in strings of CHAR16 code units. Instead, whereever necessary the semantic algorithms convert supplementary characters into pairs of *surrogate* code units before storing them into strings. The first surrogate code unit *h* is in the set {‘«uD800»’ ... ‘«uDBFF»’} and the second surrogate code unit *l* is in the set {‘«uDC00»’ ... ‘«uDFFF»’}; together they encode the supplementary character with the code point value:

$\text{0x10000} + (\mathit{char16ToInteger}(h) - \text{0xD800}) \times \text{0x400} + \mathit{char16ToInteger}(l) - \text{0xDC00}$

CHAR21 is the semantic domain of all 1114112 Unicode characters {‘«u0000»’ ... ‘«U0010FFFF»’}:

CHAR21 = CHAR16 ∪ SUPPLEMENTARYCHAR

Characters can be compared using =, ≠, <, ≤, >, and ≥. These operators compare code point values, so ‘A’ = ‘A’, ‘A’ < ‘B’, ~~and~~ ‘A’ < ‘a’, and ‘«uFFFF»’ < ‘«U00010000»’ are all **true**.

5.7.1 Character Conversions

The following procedures convert between characters and integers representing their code point or code unit values:~~procedures *characterToCode* and *codeToCharacter* convert between characters and their integer Unicode values.~~

char16ToInteger(*c*: CHAR16): {0 ... 0xFFFF}~~*characterToCode*(*c*: CHARACTER): {0 ... 65535}~~	Return the number of character *c*’s Unicode code point or code unit.~~Return character *c*’s Unicode code point as an integer~~
char21ToInteger(*c*: CHAR21): {0 ... 0x10FFFF}~~*codeToCharacter*(*i*: {0 ... 65535}): CHARACTER~~	Return the number of character *c*’s Unicode code point.~~Return the character whose Unicode code point is *i*~~
integerToChar16(*i*: {0 ... 0xFFFF}): CHAR16	Return the character whose Unicode code point or code unit number is *i*.
integerToSupplementaryChar(*i*: {0x10000 ... 0x10FFFF}): SUPPLEMENTARYCHAR	Return the character whose Unicode code point number is *i*.
integerToChar21(*i*: {0 ... 0x10FFFF}): CHAR21	Return the character whose Unicode code point number is *i*.

The procedure *digitValue* is defined as follows:

```
proc digitValue(c: {‘0’ ... ‘9’, ‘A’ ... ‘Z’, ‘a’ ... ‘z’}): {0 ... 35}
    case c of
        {‘0’ ... ‘9’} do return char16ToInteger(c) – char16ToInteger(‘0’);
        {‘A’ ... ‘Z’} do return char16ToInteger(c) – char16ToInteger(‘A’) + 10;
        {‘a’ ... ‘z’} do return char16ToInteger(c) – char16ToInteger(‘a’) + 10
    end case
end proc;
```

5.8 Lists

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

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

For example, the following list contains four strings:

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

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

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

A list can also be written using the list comprehension notation

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

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

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

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

$[x^2 \mid$ □x □ $[-1, 1, 2, 3, 4, 2, 5]] = [1, 1, 4, 9, 16, 4, 25]$
$[x{+}1 \mid$ □x □ $[-1, 1, 2, 3, 4, 5, 3, 10]$ **such that** x **mod** $2 = 1] = [0, 2, 4, 6, 4]$

Let $u = [e_0, e_1, \ldots, e_{n-1}]$ and $v = [f_0, f_1, \ldots, f_{m-1}]$ be lists, 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 $[e_i, e_{i+1}, \ldots, e_j]$ consisting of all elements of u between the i^{th} and the j^{th}, inclusive. The result is the empty list [] if $j{=}i{-}1$.
$u[i \ldots]$	$0 \le i \le \lvert u\rvert$	The list slice $[e_i, e_{i+1}, \ldots, e_{n-1}]$ consisting of all elements of u between the i^{th} and the end. The result is the empty list [] if $i{=}n$.
$u[i \setminus x]$	$0 \le i < \lvert u\rvert$	The list $[e_0, \ldots, e_{i-1}, x, e_{i+1}, \ldots, e_{n-1}]$ with the i^{th} element replaced by the value x and the other elements unchanged
$u \oplus v$		The concatenated list $[e_0, e_1, \ldots, e_{n-1}, f_0, f_1, \ldots, f_{m-1}]$
$repeat(e, i)$	$i \ge 0$	The list $[e, e, \ldots, e]$ of length i containing i identical elements e
$u = v$		**true** if the lists u and v are equal and **false** otherwise. Lists u and v are equal if they have the same length and all of their corresponding elements are equal.
$u \ne v$		**false** if the lists u and v are equal and **true** otherwise.

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

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

some x □ u **satisfies** $predicate(x)$
every x □ u **satisfies** $predicate(x)$

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

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

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

5.9 Strings

A list of CHAR16 code units~~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 section 5.1~~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 “”.

A string holds code units, not code points (see section 5.7). Supplementary Unicode characters are represented as pairs of surrogate code units when stored in strings.

In addition to the other list operations, $<$, $\le$, $>$, and $\ge$ 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 code unit~~character~~ of x is less than the first code unit~~character~~ of y, or the first code unit~~character~~ of x is equal to the first code unit~~character~~ of y and the rest of string x is less than the rest of string y.

Note that these relations compare code units, not code points, which can produce unexpected effects if a string contains supplementary characters expanded into a pairs of surrogates. For example, even though '«uFFFF»' < '«U00010000»', the supplementary character '«U00010000»' is represented in a string as "«uD800»«uDC00»", and, by the above rules, "«uFFFF»" > "«uD800»«uDC00»".

STRING is the semantic domain of all strings. STRING = ~~CHARACTER~~CHAR16[].

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

$a.\mathsf{label}_i$

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

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

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

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

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

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 $+\mathbf{zero}_{\mathbf{f32}}$ (positive zero), $-\mathbf{zero}_{\mathbf{f32}}$ (negative zero), $+\infty_{\mathbf{f32}}$ (positive infinity), $-\infty_{\mathbf{f32}}$ (negative infinity), and $\mathbf{NaN}_{\mathbf{f32}}$ (not a number).

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

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

5.12.3.1 Shorthand Notation

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

5.12.3.2 Conversion

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

```
proc realToFloat32(x: REAL): FLOAT32
    s: RATIONAL{} ← NORMALISEDFLOAT32VALUES ∪ DENORMALISEDFLOAT32VALUES ∪ {$-2^{128}$, 0, $2^{128}$};
    Let a: RATIONAL be the element of s closest to x (i.e. such that |a–x| is as small as possible). If two elements of s are
        equally close, let a be the one with an even significand; for this purpose $-2^{128}$, 0, and $2^{128}$ are considered to have
        even significands.
    if $a = 2^{128}$ then return $+\infty_{f32}$
    elsif $a = -2^{128}$ then return $-\infty_{f32}$
    elsif $a \neq 0$ then return NONZEROFINITEFLOAT32⟨⟨value: a⟩⟩
    elsif $x < 0$ then return $-\text{zero}_{f32}$
    else return $+\text{zero}_{f32}$
    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 ∈ {$+\text{zero}_{f32}$, $-\text{zero}_{f32}$} 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	$(-x.\text{value})_{f32}$
$-\text{zero}_{f32}$	$+\text{zero}_{f32}$
$+\text{zero}_{f32}$	$-\text{zero}_{f32}$
positive finite	$(-x.\text{value})_{f32}$
$+\infty_{f32}$	$-\infty_{f32}$
NaN_{f32}	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 ∪ {$+\infty_{f64}$, $-\infty_{f64}$, NaN_{f64}};
FINITEFLOAT64 = NONZEROFINITEFLOAT64 ∪ {$+\text{zero}_{f64}$, $-\text{zero}_{f64}$}

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**$_{f64}$ (positive zero), **–zero**$_{f64}$ (negative zero), **+∞**$_{f64}$ (positive infinity), **–∞**$_{f64}$ (negative infinity), and **NaN**$_{f64}$ (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**$_{f64}$ ≠ **–zero**$_{f64}$ but **NaN**$_{f64}$ = **NaN**$_{f64}$. 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_{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_{f64} is a FLOAT64 value (whose exact value is actually 3.399999999999999911182158029987476766109466552734375). The positive finite FLOAT64 values range from $(5 \times 10^{–324})_{f64}$ to $(1.7976931348623157 \times 10^{308})_{f64}$.

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 +∞$_{f64}$
    elsif a = $–2^{1024}$ then return –∞$_{f64}$
    elsif a ≠ 0 then return NONZEROFINITEFLOAT64⟨value: a⟩
    elsif x < 0 then return –zero$_{f64}$
    else return +zero$_{f64}$
    end if
end proc
```

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

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

float32ToFloat64(*x*: FLOAT32): FLOAT64

x	**Result**
–∞$_{f32}$	**–∞**$_{f64}$
–zero$_{f32}$	**–zero**$_{f64}$
+zero$_{f32}$	**+zero**$_{f64}$
+∞$_{f32}$	**+∞**$_{f64}$
NaN$_{f32}$	**NaN**$_{f64}$
Any NONZEROFINITEFLOAT32 value	NONZEROFINITEFLOAT64⟨value: *x*.value⟩

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

```
proc truncateFiniteFloat64(x: FINITEFLOAT64): INTEGER
    if x ∈ {+zero$_{f64}$, –zero$_{f64}$} 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
```

5.12.4.3 Arithmetic

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

float64Abs(*x*: FLOAT64): FLOAT64

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

float64Negate(*x*: FLOAT64): FLOAT64

x	Result
$\mathbf{-\infty}_{f64}$	$\mathbf{+\infty}_{f64}$
negative finite	$(-x.\text{value})_{f64}$
$\mathbf{-zero}_{f64}$	$\mathbf{+zero}_{f64}$
$\mathbf{+zero}_{f64}$	$\mathbf{-zero}_{f64}$
positive finite	$(-x.\text{value})_{f64}$
$\mathbf{+\infty}_{f64}$	$\mathbf{-\infty}_{f64}$
$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$

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

	y						
x	$\mathbf{-\infty}_{f64}$	negative finite	$\mathbf{-zero}_{f64}$	$\mathbf{+zero}_{f64}$	positive finite	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$
negative finite	$\mathbf{-\infty}_{f64}$	$(x.\text{value} + y.\text{value})_{f64}$	*x*	*x*	$(x.\text{value} + y.\text{value})_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{-zero}_{f64}$	$\mathbf{-\infty}_{f64}$	*y*	$\mathbf{-zero}_{f64}$	$\mathbf{+zero}_{f64}$	*y*	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{+zero}_{f64}$	$\mathbf{-\infty}_{f64}$	*y*	$\mathbf{+zero}_{f64}$	$\mathbf{+zero}_{f64}$	*y*	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
positive finite	$\mathbf{-\infty}_{f64}$	$(x.\text{value} + y.\text{value})_{f64}$	*x*	*x*	$(x.\text{value} + y.\text{value})_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$

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

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

	y						
x	$\mathbf{-\infty}_{f64}$	negative finite	$\mathbf{-zero}_{f64}$	$\mathbf{+zero}_{f64}$	positive finite	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$
negative finite	$\mathbf{+\infty}_{f64}$	$(x.\text{value} - y.\text{value})_{f64}$	*x*	*x*	$(x.\text{value} - y.\text{value})_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{-zero}_{f64}$	$\mathbf{+\infty}_{f64}$	$(-y.\text{value})_{f64}$	$\mathbf{+zero}_{f64}$	$\mathbf{-zero}_{f64}$	$(-y.\text{value})_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{+zero}_{f64}$	$\mathbf{+\infty}_{f64}$	$(-y.\text{value})_{f64}$	$\mathbf{+zero}_{f64}$	$\mathbf{+zero}_{f64}$	$(-y.\text{value})_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$
positive finite	$\mathbf{+\infty}_{f64}$	$(x.\text{value} - y.\text{value})_{f64}$	*x*	*x*	$(x.\text{value} - y.\text{value})_{f64}$	$\mathbf{-\infty}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{+\infty}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$
$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$	$\mathbf{NaN}_{f64}$

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

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

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

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

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

x \ *y*	$-\infty_{f64}$, $+\infty_{f64}$	positive or negative finite	$-\mathrm{zero}_{f64}$, $+\mathrm{zero}_{f64}$	NaN_{f64}
$-\infty_{f64}$	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}
negative finite	*x*	*float64Negate*(*float64Remainder*(*float64Negate*(*x*), *y*))	NaN_{f64}	NaN_{f64}
$-\mathrm{zero}_{f64}$	$-\mathrm{zero}_{f64}$	$-\mathrm{zero}_{f64}$	NaN_{f64}	NaN_{f64}
$+\mathrm{zero}_{f64}$	$+\mathrm{zero}_{f64}$	$+\mathrm{zero}_{f64}$	NaN_{f64}	NaN_{f64}
positive finite	*x*	(*x*.value – |*y*.value|□□*x*.value/|*y*.value|□)$_{f64}$	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 *float64Remainder*(*float64Negate*(*x*), *y*) always produces the same result as *float64Negate*(*float64Remainder*(*x*, *y*)). Also, *float64Remainder*(*x*, *float64Negate*(*y*)) always produces the same result as *float64Remainder*(*x*, *y*).

5.13 Procedures

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

A procedure is denoted as:

```
proc f(param₁: T₁, ... , paramₙ: Tₙ): T
   step₁;
   step₂;
   ... ;
   stepₘ
end proc;
```

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

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

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

5.13.1 Operations

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

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

5.13.2 Semantic Domains of Procedures

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

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* T, then exception *e* stops propagating, variable *v* is bound to the value *e*, and the second list of *step*s is performed. If *e* T, then exception *e* keeps propagating out.

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

5.13.4 Nested Procedures

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

5.14 Grammars

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

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

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

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

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

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

5.14.1 Grammar Notation

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

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

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

For example, the syntactic definition

SampleList □
  «empty»
| `...` *Identifier* (*Identifier*: 11.2)
| *SampleListPrefix*
| *SampleListPrefix* `,` `...` *Identifier*

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

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

5.14.2 Lookahead Constraints

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

For example, given the rules

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

DecimalDigits □
  DecimalDigit
| *DecimalDigits* *DecimalDigit*

the rule

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

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

5.14.3 Line Break Constraints

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

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

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

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

5.14.4 Parameterised Rules

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

Metadefinitions such as

□ □ {normal, initial}

□□ {allowIn, noIn}

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

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

expands into the following four rules:

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

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

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

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

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

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

5.14.5 Special Lexical Rules

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

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

NonAsteriskOrSlash □ *UnicodeCharacter* **except** `*` | `/`

5.15 Semantic Actions

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

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

5.15.1 Example

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

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

Digits →
      Digit
   |  Digits Digit

Numeral →
      Digits
   |  Digits # Digits
```

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

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

DecimalValue[Digits]: INTEGER;
   DecimalValue[Digits → Digit] = Value[Digit];
   DecimalValue[Digits_0 → Digits_1 Digit] = 10×DecimalValue[Digits_1] + Value[Digit];

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

Value[Numeral]: INTEGER;
   Value[Numeral → Digits] = DecimalValue[Digits];
   Value[Numeral → Digits_1 # Digits_2]
      begin
         base: INTEGER ← DecimalValue[Digits_2];
         if base ≥ 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~~ nonterminals 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.

The propagation notation is also used in when the actions return a value. In this case each expansion must have exactly one nonterminal. For example, given the grammar rule

```
Id □
    SimpleId
  | ComplexId
```

the notation

Eval[*Id*] (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME propagates the call to Eval to nonterminals in the expansion of *Id*.

is an abbreviation for the following:

```
proc Eval[Id] (env: ENVIRONMENT, phase: PHASE): MULTINAME
   [Id □  SimpleId] do return Eval[SimpleId](env, phase);
   [Id □  ComplexId] do return Eval[ComplexId](env, phase)
end proc;
```

5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

Action[*nonterminal*]: T;

This notation states that action Action can be performed on nonterminal *nonterminal* and returns a value that is a member of the semantic domain T. The action's value is either defined using the notation

Action[*nonterminal* □ *expansion*] = *expression* below or set as a side effect of computing another action via an action assignment.

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

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

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

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

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

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

```
proc Action[nonterminal □ expansion] ($param_1$: $T_1$, … , $param_n$: $T_n$): T
   $step_1$;
   $step_2$;
   … ;
   $step_m$
end proc;
```

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

```
proc Action[nonterminal] ($param_1$: $T_1$, … , $param_n$: $T_n$): T
   [nonterminal □ $expansion_1$] do
      step;
      … ;
      step;
   [nonterminal □ $expansion_2$] do
      step;
      … ;
      step;
   …;
   [nonterminal □ $expansion_n$] do
      step;
      … ;
      step
end proc;
```

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

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

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

5.16 Other Semantic Definitions

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

name: T = *expression*;

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

name: T □ *expression*;

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

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

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

6 Source Text

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

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

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

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

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

6.1 Unicode Format-Control Characters

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

The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a Unicode escape sequence (see section *****) to include a Unicode format-control character inside a string or regular expression literal.

7 Lexical Grammar

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

A *token* is one of the following:

- A keyword token, which is either:
  - One of the reserved words currently used by ECMAScript `as`, `break`, `case`, `catch`, `class`, `const`, `continue`, `default`, `delete`, `do`, `else`, `export`, `extends`, `false`, `final`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `is`, `namespace`, `new`, `null`, `package`, `private`, `public`, `return`, `static`, `super`, `switch`, `this`, `throw`, `true`, `try`, `typeof`, `use`, `var`, `void`, `while`, `with`.
  - One of the reserved words reserved for future use `abstract`, `debugger`, `enum`, `goto`, `implements`, `interface`, `native`, `protected`, `synchronized`, `throws`, `transient`, `volatile`.
  - One of the non-reserved words `exclude`, `get`, `include`, `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$^{\text{re}}$ → *WhiteSpace* *InputElement*$^{\text{re}}$ (*WhiteSpace*: 7.2)

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

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

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

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

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

Semantics

The grammar parameter □ 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$_1$] $\oplus$ [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 All digits of hexadecimal literals are significant.

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

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

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~~ CHAR16 ∪ 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~~ CHAR16 ∪ STRING;

NONPRIMITIVEOBJECT = NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ SIMPLEINSTANCE ∪ METHODCLOSURE ∪ DATE ∪ REGEXP ∪ PACKAGE;

A BINDINGOBJECT is an object that can bind local properties:

BINDINGOBJECT = CLASS ∪ SIMPLEINSTANCE ∪ REGEXP ∪ DATE ∪ PACKAGE;

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

OBJECTOPT = OBJECT ∪ {**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.

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

BOOLEANOPT = BOOLEAN ∪ {**none**};

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
category~~memberMod~~	PROPERTYCATEGORY~~MEMBERMODIFIER~~	**static**, **virtual**, or **final** if one of these attributes has been given; **none** if not. PROPERTYCATEGORY~~MEMBERMODIFIER~~ = {**none**, **static**, **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 st ~~members~~singleton properties define this class (see section *****)
instanceProperties~~instanceMembers~~	InstanceProperty~~InstanceMember~~{}	Map of qualified names to insta ~~members~~ properties defined overridden in this class
super	ClassOpt	This class's immediate superclas **null** if none
prototype	ObjectOpt	The default archetype of new insta of this class
complete	Boolean	**true** after all members of this class h been added to this Class record
name	String	This class's name
typeofString	String	A string to return if `typeof` is invo on this class's instances
privateNamespace	Namespace	This class's `private` namespace
dynamic	Boolean	**true** if this class or any of its ances was defined with the `dynamic` attrib
final	Boolean	**true** if this class cannot be subclassed
defaultValue	ObjectOpt	When a variable whose type is this c is defined but not explicitly initiali the variable's initial value defaultValue, which must be instance of this class. The class Ne has no values, so that class's (and that class's) defaultValue is **none**.
defaultHint	Hint	The default hint to use when conver an instance of this class to a primitive
hasProperty	Object □ Class □ Object □ Boolean □ Phase □ Boolean	
bracketRead	Object □ Class □ Object[] □ Boolean □ Phase □ ObjectOpt	
bracketWrite	Object □ Class □ Object[] □ Object □ Boolean □ {**run**} □ {**none**, **ok**}	
bracketDelete	Object □ Class □ Object[] □ {**run**} □ BooleanOpt	
read	Object □ Class □ Multiname □ EnvironmentOpt □ Boolean □ Phase □ ObjectOpt	
write	Object □ Class □ Multiname □ EnvironmentOpt ~~□ Boolean~~ □ Object □ Boolean □ {**run**} □ {**none**, **ok**}	
delete	Object □ Class □ Multiname □ EnvironmentOpt □ {**run**}	

	BooleanOpt	
enumerate	Object Object{}	
call	~~Object~~ Object Class Object~~{}~~[] Phase Object	A procedure to call when this class is used in a call expression. The parameters are the `this` argument, this class, the list of arguments, and the phase of evaluation (section 9.4).
construct	Class Object~~{}~~[] Phase Object	A procedure to call when this class is used in a `new` expression. The parameters are this class, the list of arguments, and the phase of evaluation (section 9.4).
init	(SimpleInstance Object[] {**run**} ()) {**none**}	A procedure to call to initialise a newly created instance of this class or **none** if no special initialisation is needed. init is called by construct.
is	Object Class Boolean	A procedure to call to determine whether a given object is an instance of this class. The parameters are the object to be tested and this class.
coerce~~implicitCoerce~~	Object Class ~~ Boolean~~ ObjectOpt~~Object~~	A procedure to call when a value is assigned to a variable, parameter, or result whose type is this class. The ~~argument~~ first parameter to ~~implicitCoerce~~ `coerce` can be any value, which may or may not be an instance of this class; the result must be an instance of this class. If the coercion is not appropriate, ~~implicitCoerce~~ coerce ~~should throw an exception if its second argument is **false** or return **null** (as long as **null** is an instance of this class) if its second argument is **true**~~returns **none**. The second parameter is this class.

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

ClassOpt = Class {**none**}

~~A Class *c* is an *ancestor* of Class *d* if either *c* = *d* or *d*.super = *s*, *s* ≠ **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
archetype~~super~~	ObjectOpt	Optional link to the next object in this instance's ~~prototype~~ archetype chain

		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 9.4) 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 9.4) 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	OBJECTOPT~~OBJECTU~~	This fixed property's current value; ~~**uninitialised**~~ **none** if the fixed property is an uninitialised constant

9.1.10 Uninstantiated Functions

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

Field	Contents	Note
type	CLASS	Values to be transferred into the generated SIMPLEINSTANCE's corresponding fields
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
slots	SLOT{}	A set of slots that hold this method closure's fixed property values

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)
archetype~~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)
archetype~~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)
archetype~~super~~	OBJECTOPT	
name	STRING	This package's name
initialize	(() □ ()) □ {**none**, **busy**}	A procedure to initialize this package
sealed	BOOLEAN	Same as in SIMPLEINSTANCEs (section 9.1.9)
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~~ Property ~~and operator~~ lookups on a LimitedInstance value will only find ~~members and operators~~properties 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.
multiname~~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.

		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 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**}

9.5 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.

9.6 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

9.7 Semantic Exceptions

All values thrown by the semantics' **throw** steps and caught by **try-catch** steps (see section 5.13.3) are members of the semantic domain SEMANTICEXCEPTION, defined as follows:

SEMANTICEXCEPTION = OBJECT □ CONTROLTRANSFER;
CONTROLTRANSFER = BREAK □ CONTINUE □ RETURN;

The semantics **throw** four different kinds of values:

- An OBJECT is thrown as a result of encountering an error or evaluating an ECMAScript `throw` statement
- A BREAK tuple is thrown as a result of evaluating an ECMAScript `break` statement
- A CONTINUE tuple is thrown as a result of evaluating an ECMAScript `continue` statement
- A RETURN tuple is thrown as a result of evaluating an ECMAScript `return` statement

A BREAK tuple (see section 5.10) has the fields below.

Field	Contents	Note
value	OBJECT	The value produced by the last statement to be executed before the `break`
label	LABEL	The label that is the target of the `break`

A CONTINUE tuple (see section 5.10) has the fields below.

Field	Contents	Note
value	OBJECT	The value produced by the last statement to be executed before the `continue`
label	LABEL	The label that is the target of the `continue`

A RETURN tuple (see section 5.10) has the field below.

Field	Contents	Note
value	OBJECT	The value of the expression in the `return` statement or **undefined** if omitted

9.8 Function Support

The FUNCTIONKIND semantic domain encodes a general kind of a function:

FUNCTIONKIND = {**plainFunction**, **uncheckedFunction**, **prototypeFunction**, **instanceFunction**, **constructorFunction**};

These kinds represent the following:

- A **plainFunction** is a static function whose signature is checked when it is called. This function is not a prototype-based constructor and cannot be used in a `new` expression.
- A **prototypeFunction** is a static function whose signature is checked when it is called. This function is also a prototype-based constructor and may be used in a `new` expression.
- An **uncheckedFunction** is a static function whose signature is not checked when it is called. This function is also a prototype-based constructor and may be used in a `new` expression.
- An **instanceFunction** is an instance method whose signature is checked when it is called.
- A **constructorFunction** is a class constructor whose signature is checked when it is called.

The subset of static function kinds has its own semantic domain STATICFUNCTIONKIND:

STATICFUNCTIONKIND = {**plainFunction**, **uncheckedFunction**, **prototypeFunction**};

Two of the above five function kinds, plain and instance functions, can be defined either normally or as getters or setters. This distinction is encoded by the HANDLING semantic domain:

HANDLING = {**normal**, **get**, **set**};

9.9 Environment~~ Frames~~s

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 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 hold the runtime values of variables and other definitions; these frames are called *instantiated frames*. Other frames, called *uninstantiated frames*, are used as templates for making (instantiating) instantiated frames. The static analysis done by Validate generates instantiated frames for a few top-level scopes and uninstantiated frames for other scopes; the *preinst* parameter to Validate governs whether it generates instantiated or uninstantiated frames.

~~9.9.1System Frame~~

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

~~**Field**~~	~~**Contents**~~	~~**Note**~~
~~localBindings~~	~~LOCALBINDING{}~~	~~Map of qualified names to definitions in this frame~~

~~9.9.2~~9.9.1 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
kind	FUNCTIONKIND	See section 9.8
handling	HANDLING	See section 9.8
callsSuperconstructor	BOOLEAN	A flag that indicates whether a call to the superclass's constructor has been detected during static analysis of a class constructor. Always **false** if kind is not **constructorFunction**.
superconstructorCalled	BOOLEAN	If kind is a **constructorFunction**, this flag indicates whether the superclass's constructor has been called yet during execution of this constructor. Always **true** if kind is not **constructorFunction**.
this	OBJECTOPT	The value of `this`; **none** if this function doesn't define `this` or it defines `this` but the value is not available because this function hasn't been called yet
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

PARAMETERFRAMEOPT consists of all parameter frames as well as **none**:

PARAMETERFRAMEOPT = PARAMETERFRAME □ {**none**};

~~9.9.2.1~~9.9.1.1 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

~~9.9.3~~9.9.2 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 field below.

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

~~9.9.4~~9.9.3 With Frames

Frames holding bindings for `with` statements are represented as WithFrame records (see section 5.11) with the field below.

Field	Contents	Note
value	ObjectOpt	The value of the with statement's expression or **none** if not evaluated yet

9.10 Environment Bindings

In general, accesses of ~~members~~ properties 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~~ properties are visible only for read or only for write accesses; other ~~members~~ properties are visible to both read and write accesses. The tag **readWrite** indicates that a ~~member~~ property 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~~ properties generally have the access **readWrite** but an attempt to write one results in an error. Trying to write to ~~member~~ a property with the access **read** would not even find the ~~member~~property, and the write would proceed to search an object's parent hierarchy for another matching ~~member~~property.

PropertyOpt = SingletonProperty □ InstanceProperty □ {**none**};

9.10.1 Static Bindings

A LocalBinding tuple (see section 5.10) has the fields below and describes the ~~member~~ property to which one qualified name is bound in a frame. Multiple qualified names may be bound to the same ~~member~~ property in a frame, but a qualified name may not be bound to multiple ~~members~~ properties 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~~ property is visible
explicit	Boolean	**true** if this binding should not be imported into the global scope
enumerable	Boolean	**true** if this binding should be visible in a `for-in` statement
content	SingletonProperty~~LocalMember~~	The ~~member~~ property to which this qualified name was bound

A ~~local member~~singleton property is a property that is not an instance property. A singleton property is either **forbidden**, a variable, a dynamic variable, a getter, or a setter:

SingletonProperty~~LocalMember~~ = {**forbidden**} □ Variable □ DynamicVar □ Getter □ Setter;

SingletonPropertyOpt~~LocalMemberOpt~~ = SingletonProperty~~LocalMember~~ □ {**none**};

A **forbidden** ~~static member~~singleton property 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	CLASS	Type of values that may be stored in this variable
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.
initializer	INITIALIZER □ {**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.
initializerEnv	ENVIRONMENT	The environment to provide to initializer 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 value can be either an object, **none** (used when the variable has not been initialised yet), or an uninstantiated function (compile time only):

VARIABLEVALUE = {**none**} □ OBJECT □ UNINSTANTIATEDFUNCTION;

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

INITIALIZER = ENVIRONMENT □ PHASE □ OBJECT;

INITIALIZEROPT = INITIALIZER □ {**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 GETTER record (see section 5.11) has the fields below and describes one static getter definition.

Field	Contents	Note
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	ENVIRONMENTOPT	The environment bound to this getter; **none** if not yet instantiated

A SETTER record (see section 5.11) has the fields below and describes one static setter definition.

Field	Contents	Note
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	EnvironmentOpt	The environment bound to this setter; **none** if not yet instantiated

9.10.2 Instance Bindings

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

InstanceProperty~~InstanceMember~~ = InstanceVariable ∪ InstanceMethod ∪ InstanceGetter ∪ InstanceSetter;

InstanceProperty~~InstanceMember~~Opt = InstanceProperty~~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	ObjectOpt	This variable's default value; **none** if not 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 encoded in the ParameterFrame's parameters, rest, and returnType fields
length	Integer	The instance method's preferred number of arguments
call	Object × Object[] × Phase → Object	A procedure to call when this instance method is invoked. The procedure takes a `this` Object, a list of argument Objects, and a Phase (see section 9.4) and produces an Object result

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

Field	Contents	Note
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
signature	ParameterFrame	This getter's signature encoded in the ParameterFrame's parameters, rest, and returnType fields
call	Object × Phase → Object	A procedure to call to read the value, passing it the `this` value and the current mode of expression evaluation

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
signature	PARAMETERFRAME	This setter's signature encoded in the PARAMETERFRAME's parameters, rest, and returnType fields
call	OBJECT × OBJECT × PHASE → ()	A procedure to call to write the value, passing it the `this` value, the value being written, and the current mode of expression evaluation

9.11 Miscellaneous

~~9.11~~9.11.1 Extended Integers and Rationals

An extended integer is an integer or one of the tags **+∞**, **–∞**, or **NaN**:

EXTENDEDINTEGER = INTEGER ∪ {**+∞**, **–∞**, **NaN**};

An extended rational is a rational number with 0 replaced by the tags **+zero** and **–zero** or one of the tags **+∞**, **–∞**, or **NaN**:

EXTENDEDRATIONAL = (RATIONAL – {0}) ∪ {**+zero**, **–zero**, **+∞**, **–∞**, **NaN**};

9.11.2 Order

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

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

9.11.3 Hints

A hint describes whether converting an object to a primitive should favour conversions to strings (**hintString**) or to numbers (**hintNumber**).

HINT = {**hintString**, **hintNumber**};
HINTOPT = HINT ∪ {**none**};

10 Data Operations

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

10.1 Numeric Utilities

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

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

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

```
proc signedWrap32(i: INTEGER): $\{-2^{31} \ldots 2^{31} - 1\}$
   j: INTEGER $\leftarrow$ 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 $\leftarrow$ bitwiseAnd(i, 0xFFFFFFFFFFFFFFFF);
   if $j \geq 2^{63}$ then j $\leftarrow$ $j - 2^{64}$ end if;
   return j
end proc;
```

truncateToInteger(*x*) returns *x* converted to an integer by rounding towards zero. If *x* is an infinity or a NaN, the result is 0.

```
proc truncateToInteger(x: GENERALNUMBER): INTEGER
   case x of
      {NaN_f32, NaN_f64, +∞_f32, +∞_f64, –∞_f32, –∞_f64} do return 0;
      FINITEFLOAT32 do return truncateFiniteFloat32(x);
      FINITEFLOAT64 do return truncateFiniteFloat64(x);
      LONG ∪ ULONG do return x.value
   end case
end proc;
```

pinExtendedInteger(*i*, *limit*, *negativeFromEnd*) returns *i* pinned to the set {0 ... *limit*}, where *limit* is a nonnegative integer. If *negativeFromEnd* is **true**, then negative values of *i* from –*limit* through –1 are treated as 0 through *limit* – 1 respectively.

```
proc pinExtendedInteger(i: EXTENDEDINTEGER, limit: INTEGER, negativeFromEnd: BOOLEAN): INTEGER
   case i of
      {NaN} do throw a RangeError exception;
      {–∞} do return 0;
      {+∞} do return limit;
      INTEGER do
         j: INTEGER $\leftarrow$ i;
         if $j > limit$ then j $\leftarrow$ limit end if;
         if negativeFromEnd and $j < 0$ then j $\leftarrow$ $j + limit$ end if;
         if $j < 0$ then j $\leftarrow$ 0 end if;
         note  $0 \leq j \leq limit$;
         return j
   end case
end proc;
```

checkInteger(*x*) returns *x* converted to an integer if its mathematical value is, in fact, an integer. If *x* is an infinity or a NaN or has a fractional part, the result is **none**.

```
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;
```

integerToLong(*i*) converts *i* to the first of the types Long, ULong, or Float64 that can contain the value *i*. If necessary, the Float64 result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

```
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 $i_{f64}$
   end if
end proc;
```

integerToULong(*i*) converts *i* to the first of the types ULong, Long, or Float64 that can contain the value *i*. If necessary, the Float64 result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

```
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 $i_{f64}$
   end if
end proc;
```

rationalToLong(*q*) converts *q* to one of the types Long, ULong, or Float64, whichever one can come the closest to representing the true value of *q*. If several of these types can come equally close to the value of *q*, then one of them is chosen according to the algorithm below.

```
proc rationalToLong(q: Rational): GeneralNumber
   if q □ Integer then return integerToLong(q)
   elsif $|q| \le 2^{53}$ then return $q_{f64}$
   elsif $q < -2^{63} - 1/2$ or $q \ge 2^{64} - 1/2$ then return $q_{f64}$
   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;
```

rationalToULong(*q*) converts *q* to one of the types ULong, Long, or Float64, whichever one can come the closest to representing the true value of *q*. If several of these types can come equally close to the value of *q*, then one of them is chosen according to the algorithm below.

```
proc rationalToULong(q: Rational): GeneralNumber
   if q □ Integer then return integerToULong(q)
   elsif $|q| \le 2^{53}$ then return $q_{f64}$
   elsif $q < -2^{63} - 1/2$ or $q \ge 2^{64} - 1/2$ then return $q_{f64}$
   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 extendedRationalToFloat32(q: EXTENDEDRATIONAL): FLOAT32
    case q of
        RATIONAL do return $q_{f32}$;
        {+zero} do return $+zero_{f32}$;
        {–zero} do return $–zero_{f32}$;
        {+∞} do return $+∞_{f32}$;
        {–∞} do return $–∞_{f32}$;
        {NaN} do return $NaN_{f32}$
    end case
end proc;
```

```
proc extendedRationalToFloat64(q: EXTENDEDRATIONAL): FLOAT64
    case q of
        RATIONAL do return $q_{f64}$;
        {+zero} do return $+zero_{f64}$;
        {–zero} do return $–zero_{f64}$;
        {+∞} do return $+∞_{f64}$;
        {–∞} do return $–∞_{f64}$;
        {NaN} do return $NaN_{f64}$
    end case
end proc;
```

toRational(*x*) returns the exact RATIONAL value of *x*.

```
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;
```

toFloat32(*x*) converts *x* to a FLOAT32, using the IEEE 754 “round to nearest” mode.

```
proc toFloat32(x: GENERALNUMBER): FLOAT32
    case x of
        LONG □ ULONG do return $(x.value)_{f32}$;
        FLOAT32 do return x;
        {$–∞_{f64}$} do return $–∞_{f32}$;
        {$–zero_{f64}$} do return $–zero_{f32}$;
        {$+zero_{f64}$} do return $+zero_{f32}$;
        {$+∞_{f64}$} do return $+∞_{f32}$;
        {$NaN_{f64}$} do return $NaN_{f32}$;
        NONZEROFINITEFLOAT64 do return $(x.value)_{f32}$
    end case
end proc;
```

toFloat64(*x*) converts *x* to a FLOAT64, using the IEEE 754 “round to nearest” mode.

```
proc toFloat64(x: GENERALNUMBER): FLOAT64
    case x of
        LONG □ ULONG do return $(x.value)_{f64}$;
        FLOAT32 do return float32ToFloat64(x);
        FLOAT64 do return x
    end case
end proc;
```

generalNumberCompare(*x*, *y*) compares *x* with *y* using the IEEE 754 rules and returns **less** if $x<y$, **equal** if $x=y$, **greater** if $x>y$, or **unordered** if either *x* or *y* is a NaN. The comparison is done using the exact values of *x* and *y*, even if they have different types. Positive infinities compare equal to each other and greater than any other non-NaN values. Negative infinities compare equal to each other and less than any other non-NaN values. Positive and negative zeroes compare equal to each other.

```
proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
   if x □ {NaN$_{f32}$, NaN$_{f64}$} or y □ {NaN$_{f32}$, NaN$_{f64}$} then return unordered
   elsif x □ {+∞$_{f32}$, +∞$_{f64}$} and y □ {+∞$_{f32}$, +∞$_{f64}$} then return equal
   elsif x □ {–∞$_{f32}$, –∞$_{f64}$} and y □ {–∞$_{f32}$, –∞$_{f64}$} then return equal
   elsif x □ {+∞$_{f32}$, +∞$_{f64}$} or y □ {–∞$_{f32}$, –∞$_{f64}$} then return greater
   elsif x □ {–∞$_{f32}$, –∞$_{f64}$} or y □ {+∞$_{f32}$, +∞$_{f64}$} then return less
   else
      xr: RATIONAL □ toRational(x);
      yr: RATIONAL □ toRational(y);
      if $xr < yr$ then return less
      elsif $xr > yr$ then return greater
      else return equal
      end if
   end if
end proc;
```

10.2 Character Utilities

```
proc integerToUTF16(i: {0 ... 0x10FFFF}): STRING
   if $0 \le i \le$ 0xFFFF then return [integerToChar16(i)]
   else
      j: {0 ... 0xFFFFF} □ i – 0x10000;
      high: CHAR16 □ integerToChar16(0xD800 + bitwiseShift(j, –10));
      low: CHAR16 □ integerToChar16(0xDC00 + bitwiseAnd(j, 0x3FF));
      return [high, low]
   end if
end proc;

proc char21ToUTF16(ch: CHAR21): STRING
   return integerToUTF16(char21ToInteger(ch))
end proc;

proc surrogatePairToSupplementaryChar(h: CHAR16, l: CHAR16): SUPPLEMENTARYCHAR
   codePoint: {0x10000 ... 0x10FFFF} □
         0x10000 + (char16ToInteger(h) – 0xD800)□0x400 + char16ToInteger(l) – 0xDC00;
   return integerToSupplementaryChar(codePoint)
end proc;

proc stringToUTF32(s: STRING): CHAR21[]
   i: INTEGER □ 0;
   result: CHAR21[] □ [];
   while $i \ne |s|$ do
      ch: CHAR21;
      if s[i] □ {‘«uD800»’ ... ‘«uDBFF»’} and $i + 1 \ne |s|$ and s[i + 1] □ {‘«uDC00»’ ... ‘«uDFFF»’} then
         ch □ surrogatePairToSupplementaryChar(s[i], s[i + 1]);
         i □ i + 2
      else ch □ s[i]; i □ i + 1
      end if;
      result □ result ⊕ [ch]
   end while;
   return result
end proc;
```

```
proc charToLowerFull(ch: CHAR21): STRING
    return ch converted to a lower case character using the Unicode full, locale-independent case mapping. A single
    character may be converted to multiple characters. If ch has no lower case equivalent, then the result is the string
    char21ToUTF16(ch).
end proc;

proc charToLowerLocalized(ch: CHAR21): STRING
    return ch converted to a lower case character using the Unicode full case mapping in the host environment's current
    locale. A single character may be converted to multiple characters. If ch has no lower case equivalent, then the result is
    the string char21ToUTF16(ch).
end proc;

proc charToUpperFull(ch: CHAR21): STRING
    return ch converted to a upper case character using the Unicode full, locale-independent case mapping. A single
    character may be converted to multiple characters. If ch has no upper case equivalent, then the result is the string
    char21ToUTF16(ch).
end proc;

proc charToUpperLocalized(ch: CHAR21): STRING
    return ch converted to a upper case character using the Unicode full case mapping in the host environment's current
    locale. A single character may be converted to multiple characters. If ch has no upper case equivalent, then the result is
    the string char21ToUTF16(ch).
end proc;
```

10.3 Object Utilities

10.3.1 Object Class Inquiries

objectType(*o*) returns an OBJECT *o*'s most specific type. Although *objectType* is used internally throughout this specification, in order to allow one programmer-visible class to be implemented as an ensemble of implementation-specific classes, no way is provided for a user program to directly obtain the result of calling *objectType* on an object.

```
proc objectType(o: OBJECT): CLASS
    case o of
        UNDEFINED do return Void;
        NULL do return Null;
        BOOLEAN do return Boolean;
        LONG do return long;
        ULONG do return ulong;
        FLOAT32 do return float;
        FLOAT64 do return Number;
        CHAR16 do return char;
        STRING do return String;
        NAMESPACE do return Namespace;
        COMPOUNDATTRIBUTE do return Attribute;
        CLASS do return Class;
        SIMPLEINSTANCE do return o.type;
        METHODCLOSURE do return Function;
        DATE do return Date;
        REGEXP do return RegExp;
        PACKAGE do return Package
    end case
end proc;
```

is(*o*, *c*) returns **true** if *o* is an instance of class *c* or one of its subclasses.

```
proc is(o: OBJECT, c: CLASS): BOOLEAN
    return c.is(o, c)
end proc;
```

ordinaryIs(*o*, *c*) is the implementation of *is* for a native class unless specified otherwise in the class's definition. Host classes may either also use *ordinaryIs* or define a different procedure to perform this test.

```
proc ordinaryIs(o: OBJECT, c: CLASS): BOOLEAN
    return isAncestor(c, objectType(o))
end proc;
```

Return an ordered list of class *c*'s ancestors, including *c* itself.

```
proc ancestors(c: CLASS): CLASS[]
    s: CLASSOPT □ c.super;
    if s = none then return [c] else return ancestors(s) ⊕ [c] end if
end proc;
```

Return **true** if *c* is *d* or an ancestor of *d*.

```
proc isAncestor(c: CLASS, d: CLASS): BOOLEAN
    if c = d then return true
    else
        s: CLASSOPT □ d.super;
        if s = none then return false end if;
        return isAncestor(c, s)
    end if
end proc;
```

10.3.2 Object to Boolean Conversion

objectToBoolean(*o*) returns *o* converted to a *Boolean*.

```
proc objectToBoolean(o: OBJECT): 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 ≠ “”;
        CHAR16 □ NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ SIMPLEINSTANCE □ METHODCLOSURE □ DATE □
                REGEXP □ PACKAGE do
            return true
    end case
end proc;
```

10.3.3 Object to Primitive Conversion

```
proc objectToPrimitive(o: OBJECT, hint: HINTOPT, phase: PHASE): PRIMITIVEOBJECT
   if o ∈ PRIMITIVEOBJECT then return o end if;
   c: CLASS ← objectType(o);
   h: HINT;
   if hint ∈ HINT then h ← hint else h ← c.defaultHint end if;
   case h of
      {hintString} do
         toStringMethod: OBJECTOPT ← c.read(o, c, {public::“toString”}, none, false, phase);
         if toStringMethod ≠ none then
            r: OBJECT ← call(o, toStringMethod, [], phase);
            if r ∈ PRIMITIVEOBJECT then return r end if
         end if;
         valueOfMethod: OBJECTOPT ← c.read(o, c, {public::“valueOf”}, none, false, phase);
         if valueOfMethod ≠ none then
            r: OBJECT ← call(o, valueOfMethod, [], phase);
            if r ∈ PRIMITIVEOBJECT then return r end if
         end if;
      {hintNumber} do
         valueOfMethod: OBJECTOPT ← c.read(o, c, {public::“valueOf”}, none, false, phase);
         if valueOfMethod ≠ none then
            r: OBJECT ← call(o, valueOfMethod, [], phase);
            if r ∈ PRIMITIVEOBJECT then return r end if
         end if;
         toStringMethod: OBJECTOPT ← c.read(o, c, {public::“toString”}, none, false, phase);
         if toStringMethod ≠ none then
            r: OBJECT ← call(o, toStringMethod, [], phase);
            if r ∈ PRIMITIVEOBJECT then return r end if
         end if
   end case;
   throw a TypeError exception — cannot convert this object to a primitive
end proc;
```

10.3.4 Object to Number Conversions

objectToGeneralNumber(*o*, *phase*) returns *o* converted to a *GeneralNumber*. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
   a: PRIMITIVEOBJECT ← objectToPrimitive(o, hintNumber, phase);
   case a of
      UNDEFINED do return $NaN_{f64}$;
      NULL ∪ {false} do return $+zero_{f64}$;
      {true} do return $1_{f64}$;
      GENERALNUMBER do return a;
      CHAR16 ∪ STRING do return stringToFloat64(toString(a))
   end case
end proc;
```

objectToFloat32(*o*, *phase*) returns *o* converted to a FLOAT32. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToFloat32(o: OBJECT, phase: PHASE): FLOAT32
   a: PRIMITIVEOBJECT ← objectToPrimitive(o, hintNumber, phase);
   case a of
      UNDEFINED do return $\mathrm{NaN}_{f32}$;
      NULL ∪ {false} do return $+\mathrm{zero}_{f32}$;
      {true} do return $1_{f32}$;
      GENERALNUMBER do return toFloat32(a);
      CHAR16 ∪ STRING do return stringToFloat32(toString(a))
   end case
end proc;
```

objectToFloat64(*o*, *phase*) returns *o* converted to a FLOAT64. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToFloat64(o: OBJECT, phase: PHASE): FLOAT64
   return toFloat64(objectToGeneralNumber(o, phase))
end proc;
```

objectToExtendedInteger(*o*, *phase*) returns *o* converted to an EXTENDEDINTEGER. An error occurs if *o* has a fractional part or is a NaN. If *o* is a string, then it is converted exactly. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToExtendedInteger(o: OBJECT, phase: PHASE): EXTENDEDINTEGER
   a: PRIMITIVEOBJECT ← objectToPrimitive(o, hintNumber, phase);
   case a of
      NULL ∪ {false} do return 0;
      {true} do return 1;
      {undefined, $\mathrm{NaN}_{f32}$, $\mathrm{NaN}_{f64}$} do return NaN;
      {$+\infty_{f32}$, $+\infty_{f64}$} do return $+\infty$;
      {$-\infty_{f32}$, $-\infty_{f64}$} do return $-\infty$;
      {$+\mathrm{zero}_{f32}$, $+\mathrm{zero}_{f64}$, $-\mathrm{zero}_{f32}$, $-\mathrm{zero}_{f64}$} do return 0;
      LONG ∪ ULONG do return a.value;
      NONZEROFINITEFLOAT32 ∪ NONZEROFINITEFLOAT64 do
         r: RATIONAL ← a.value;
         if r ∉ INTEGER then
            throw a RangeError exception — the value a is not an integer
         end if;
         return r;
      CHAR16 ∪ STRING do return stringToExtendedInteger(toString(a))
   end case
end proc;
```

objectToInteger(*o*, *phase*) returns *o* converted to an INTEGER. An error occurs if *o* has a fractional part or is not finite. If *o* is a string, then it is converted exactly. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToInteger(o: OBJECT, phase: PHASE): INTEGER
   i: EXTENDEDINTEGER ← objectToExtendedInteger(o, phase);
   case i of
      {$+\infty$, $-\infty$, NaN} do throw a RangeError exception — i is not an integer;
      INTEGER do return i
   end case
end proc;
```

```
proc stringToFloat32(s: STRING): FLOAT32
    Apply the lexer grammar with the start symbol StringNumericLiteral to the string s.
    if the grammar cannot interpret the entire string as an expansion of StringNumericLiteral then
        return $\mathrm{NaN}_{f32}$
    else
        q: EXTENDEDRATIONAL □ the value of the action Lex applied to the obtained expansion of the nonterminal
        StringNumericLiteral;
        return extendedRationalToFloat32(q)
    end if
end proc;

proc stringToFloat64(s: STRING): FLOAT64
    Apply the lexer grammar with the start symbol StringNumericLiteral to the string s.
    if the grammar cannot interpret the entire string as an expansion of StringNumericLiteral then
        return $\mathrm{NaN}_{f64}$
    else
        q: EXTENDEDRATIONAL □ the value of the action Lex applied to the obtained expansion of the nonterminal
        StringNumericLiteral;
        return extendedRationalToFloat64(q)
    end if
end proc;

proc stringToExtendedInteger(s: STRING): EXTENDEDINTEGER
    Apply the lexer grammar with the start symbol StringNumericLiteral to the string s.
    if the grammar cannot interpret the entire string as an expansion of StringNumericLiteral then
        throw a TypeError exception — the string s does not contain a number
    else
        q: EXTENDEDRATIONAL □ the value of the action Lex applied to the obtained expansion of the nonterminal
        StringNumericLiteral;
        case q of
            {+zero, –zero} do return 0;
            {+∞, –∞, NaN} do return q;
            RATIONAL do
                if q □ INTEGER then return q
                else throw a RangeError exception — the value should be an integer
                end if
        end case
    end if
end proc;
```

10.3.5 Object to String Conversions

objectToString(*o*, *phase*) returns *o* converted to a *String*. If *phase* is **compile**, only constant conversions are permitted.

```
proc objectToString(o: OBJECT, phase: PHASE): STRING
    a: PRIMITIVEOBJECT □ objectToPrimitive(o, hintString, phase);
    case a of
        UNDEFINED do return “undefined”;
        NULL do return “null”;
        {false} do return “false”;
        {true} do return “true”;
        GENERALNUMBER do return generalNumberToString(a);
        CHAR16 do return [a];
        STRING do return a
    end case
end proc;
```

```
proc toString(o: Char16 ∪ String): String
    case o of
        Char16 do return [o];
        String do return o
    end case
end proc;

proc generalNumberToString(x: GeneralNumber): String
    case x of
        Long ∪ ULong do return integerToString(x.value);
        Float32 do return float32ToString(x);
        Float64 do return float64ToString(x)
    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 ← $\lfloor i/10 \rfloor$;
    r: Integer ← $i – q \cdot 10$;
    c: Char16 ← integerToChar16(r + char16ToInteger(‘0’));
    if $q = 0$ then return [c] else return integerToString(q) ⊕ [c] end if
end proc;
```

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

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

proc exponentialNotationString(digits: String, e: Integer): String
    mantissa: String;
    if $|digits| = 1$ then mantissa ← digits
    else mantissa ← [digits[0]] ⊕ “.” ⊕ digits[1 ...]
    end if;
    return mantissa ⊕ “e” ⊕ integerToStringWithSign(e)
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
      {NaN_f32} do return "NaN";
      {+zero_f32, –zero_f32} do return "0";
      {+∞_f32} do return "Infinity";
      {–∞_f32} do return "-Infinity";
      NONZEROFINITEFLOAT32 do
         r: RATIONAL ← x.value;
         if r < 0 then return "-" ⊕ float32ToString(float32Negate(x))
         else
            Let e, k, and s be integers such that $k \ge 1$, $10^{k-1} \le s \le 10^k$, $(s \times 10^{e+1-k})_{f32} = x$, and k is as small as possible.
            note  k is the number of digits in the decimal representation of s, s is not divisible by 10, and the least
                  significant digit of s is not necessarily uniquely determined by the above criteria.
            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 rule: Select the value of s for which $s \times 10^{e+1-k}$ 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 $k – 1 \le e \le 20$ then return digits ⊕ repeat('0', e + 1 – k)
            elsif $0 \le e \le 20$ then return digits[0 ... e] ⊕ "." ⊕ digits[e + 1 ...]
            elsif $–6 \le e < 0$ then return "0." ⊕ repeat('0', –(e + 1)) ⊕ digits
            else return exponentialNotationString(digits, e)
            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
      {NaN_f64} do return "NaN";
      {+zero_f64, –zero_f64} do return "0";
      {+∞_f64} do return "Infinity";
      {–∞_f64} do return "-Infinity";
      NONZEROFINITEFLOAT64 do
         r: RATIONAL ← x.value;
         if r < 0 then return "-" ⊕ float64ToString(float64Negate(x))
         else
            Let e, k, and s be integers such that $k \ge 1$, $10^{k-1} \le s \le 10^k$, $(s \times 10^{e+1-k})_{f64} = x$, and k is as small as possible.
            note  k is the number of digits in the decimal representation of s, s is not divisible by 10, and the least
                  significant digit of s is not necessarily uniquely determined by the above criteria.
            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 rule: Select the value of s for which $s \times 10^{e+1-k}$ 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 $k – 1 \le e \le 20$ then return digits ⊕ repeat('0', e + 1 – k)
            elsif $0 \le e \le 20$ then return digits[0 ... e] ⊕ "." ⊕ digits[e + 1 ...]
            elsif $–6 \le e < 0$ then return "0." ⊕ repeat('0', –(e + 1)) ⊕ digits
            else return exponentialNotationString(digits, e)
            end if
         end if
   end case
end proc;
```

10.3.6 Object to Qualified Name Conversion

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

```
proc objectToQualifiedName(o: OBJECT, phase: PHASE): QUALIFIEDNAME
    return public::(objectToString(o, phase))
end proc;
```

10.3.7 Object to Class Conversion

objectToClass(*o*) returns *o* converted to a non-**null** *Class*.

```
proc objectToClass(o: OBJECT): CLASS
    if o ∈ CLASS then return o else throw a TypeError exception end if
end proc;
```

10.3.8 Object to Attribute Conversion

objectToAttribute(*o*) returns *o* converted to an attribute.

```
proc objectToAttribute(o: OBJECT, phase: PHASE): ATTRIBUTE
    if o ∈ ATTRIBUTE then return o
    else
        note  If o is not an attribute, try to call it with no arguments.
        a: OBJECT ← call(null, o, [], phase);
        if a ∈ ATTRIBUTE then return a else throw a TypeError exception end if
    end if
end proc;
```

10.3.9 Implicit Coercions

coerce(*o*, *c*) attempts to implicitly coerce *o* to class *c*. If the coercion succeeds, *coerce* returns the coerced value. If not, *coerce* throws a *TypeError*.

The coercion always succeeds and returns *o* unchanged if *o* is already a member of class *c*. The value returned from *coerce* always is a member of class *c*.

```
proc coerce(o: OBJECT, c: CLASS): OBJECT
    result: OBJECTOPT ← c.coerce(o, c);
    if result ≠ none then return result
    else throw a TypeError exception — coercion failed
    end if
end proc;
```

coerceOrNull(*o*, *c*) attempts to implicitly coerce *o* to class *c*. If the coercion succeeds, *coerceOrNull* returns the coerced value. If not, then *coerceOrNull* returns **null** if **null** is a member of type *c*; otherwise, *coerceOrNull* throws a *TypeError*.

The coercion always succeeds and returns *o* unchanged if *o* is already a member of class *c*. The value returned from *coerceOrNull* always is a member of class *c*.

```
proc coerceOrNull(o: OBJECT, c: CLASS): OBJECT
    result: OBJECTOPT ← c.coerce(o, c);
    if result ≠ none then return result
    elsif c.coerce(null, c) = null then return null
    else throw a TypeError exception — coercion failed
    end if
end proc;
```

coerceNonNull(*o*, *c*) attempts to implicitly coerce *o* to class *c*. If the coercion succeeds and the result is not **null**, then *coerceNonNull* returns the coerced value. If not, *coerceNonNull* throws a *TypeError*.

```
proc coerceNonNull(o: OBJECT, c: CLASS): OBJECT
   result: OBJECTOPT ← c.coerce(o, c);
   if result ∉ {none, null} then return result
   else throw a TypeError exception — coercion failed
   end if
end proc;
```

ordinaryCoerce(*o*, *c*) is the implementation of coercion for a native class unless specified otherwise in the class's definition. Host classes may define a different procedure to perform this coercion.

```
proc ordinaryCoerce(o: OBJECT, c: CLASS): OBJECTOPT
   if o = null or is(o, c) then return o else return none end if
end proc;
```

10.3.10 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,
               category: 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.
      if (a.category ≠ none and b.category ≠ none and a.category ≠ b.category) or (a.overrideMod ≠ none and
            b.overrideMod ≠ none and a.overrideMod ≠ b.overrideMod) then
         throw an AttributeError exception — attributes a and b have conflicting contents
      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, category: a.category ≠ none ? a.category : b.category,
               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,
               category: none, overrideMod: none, prototype: false, unused: false⟩⟩
      NAMESPACE do
         return COMPOUNDATTRIBUTE⟨⟨namespaces: {a}, explicit: false, enumerable: false, dynamic: false,
               category: none, overrideMod: none, prototype: false, unused: false⟩⟩
      COMPOUNDATTRIBUTE do return a
   end case
end proc;
```

10.4 Access Utilities

accessesOverlap(*accesses1*, *accesses2*) returns **true** if the two ACCESSSETs have a nonempty intersection.

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

proc archetype(o: Object): ObjectOpt
    case o of
        Undefined □ Null do return none;
        Boolean do return Boolean.prototype;
        Long do return long.prototype;
        ULong do return ulong.prototype;
        Float32 do return float.prototype;
        Float64 do return Number.prototype;
        Char16 do return char.prototype;
        String do return String.prototype;
        Namespace do return Namespace.prototype;
        CompoundAttribute do return Attribute.prototype;
        MethodClosure do return Function.prototype;
        Class do return Class.prototype;
        SimpleInstance □ RegExp □ Date □ Package do return o.archetype
    end case
end proc;
```

archetypes(*o*) returns the set of *o*'s archetypes, not including *o* itself.

```
proc archetypes(o: Object): Object{}
    a: ObjectOpt □ archetype(o);
    if a = none then return {} end if;
    return {a} □ archetypes(a)
end proc;
```

o is an object that is known to have slot *id*. *findSlot*(*o*, *id*) returns that slot.

```
proc findSlot(o: Object, id: InstanceVariable): Slot
    note o must be a SimpleInstance or a MethodClosure in order to have slots.
    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 □ Object end if;
            v.type □ type;
            v.setup □ none;
        {none} do nothing;
        {busy} do
            throw a ConstantError exception — a constant's type or initialiser cannot depend on the value of that constant
    end case
end proc;
```

writeVariable(*v*, *newValue*, *clearInitializer*) writes the value *newValue* into the mutable or immutable variable *v*. *newValue* is coerced to *v*'s type. If the *clearInitializer* flag is set, then the caller has just evaluated *v*'s initialiser and is supplying its result in *newValue*. In this case *writeVariable* atomically clears *v*.initializer while writing *v*.value. In all other cases the presence of an initialiser or an existing value will prevent an immutable variable's value from being written.

```
proc writeVariable(v: VARIABLE, newValue: OBJECT, clearInitializer: BOOLEAN): OBJECT
    coercedValue: OBJECT ← coerce(newValue, v.type);
    if clearInitializer then v.initializer ← none end if;
    if v.immutable and (v.value ≠ none or v.initializer ≠ none) then
        throw a ReferenceError exception — cannot initialise a const variable twice
    end if;
    v.value ← coercedValue;
    return coercedValue
end proc;
```

10.5 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;
```

If *env* is from within a function's body, *getEnclosingParameterFrame*(*env*) returns the PARAMETERFRAME for the innermost such function; otherwise, it returns **none**.

```
proc getEnclosingParameterFrame(env: ENVIRONMENT): PARAMETERFRAMEOPT
    for each frame ∈ env do
        case frame of
            LOCALFRAME ∪ WITHFRAME do nothing;
            PARAMETERFRAME do return frame;
            PACKAGE ∪ CLASS do return none
        end case
    end for each;
    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;
```

getPackageFrame(*env*) returns the innermost package frame in *env*.

```
proc getPackageFrame(env: ENVIRONMENT): PACKAGE
   i: INTEGER ← 0;
   while env[i] ∉ PACKAGE do i ← i + 1 end while;
   note Every environment ends with a PACKAGE frame, so one will always be found.
   return env[i]
end proc;
```

10.6 Property Lookup

findLocalSingletonProperty(*o*, *multiname*, *access*) looks in *o* for a local singleton property with one of the names in *multiname* and access that includes *access*. If there is no such property, *findLocalSingletonProperty* returns **none**. If there is exactly one such property, *findLocalSingletonProperty* returns it. If there is more than one such property, *findLocalSingletonProperty* throws an error.

```
proc findLocalSingletonProperty(o: NONWITHFRAME ∪ SIMPLEINSTANCE ∪ REGEXP ∪ DATE, multiname: MULTINAME,
      access: ACCESS): SINGLETONPROPERTYOPT
   matchingLocalBindings: LOCALBINDING{} ← {b | ∀b ∈ o.localBindings such that
         b.qname ∈ multiname and accessesOverlap(b.accesses, access)};
   note If the same property was found via several different bindings b, then it will appear only once in the set
         matchingProperties.
   matchingProperties: SINGLETONPROPERTY{} ← {b.content | ∀b ∈ matchingLocalBindings};
   if matchingProperties = {} then return none
   elsif |matchingProperties| = 1 then return the one element of matchingProperties
   else
      throw a ReferenceError exception — this access is ambiguous because the bindings it found belong to several
            different local properties
   end if
end proc;
```

instancePropertyAccesses(*m*) returns instance property's ACCESSSET.

```
proc instancePropertyAccesses(m: INSTANCEPROPERTY): ACCESSSET
   case m of
      INSTANCEVARIABLE ∪ INSTANCEMETHOD do return readWrite;
      INSTANCEGETTER do return read;
      INSTANCESETTER do return write
   end case
end proc;
```

findLocalInstanceProperty(*c*, *multiname*, *accesses*) looks in class *c* for a local instance property with one of the names in *multiname* and accesses that have a nonempty intersection with *accesses*. If there is no such property, *findLocalInstanceProperty* returns **none**. If there is exactly one such property, *findLocalInstanceProperty* returns it. If there is more than one such property, *findLocalInstanceProperty* throws an error.

```
proc findLocalInstanceProperty(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
   matches: INSTANCEPROPERTY{} ← {m | ∀m ∈ c.instanceProperties such that m.multiname ∩ multiname ≠ {} and
         accessesOverlap(instancePropertyAccesses(m), accesses)};
   if matches = {} then return none
   elsif |matches| = 1 then return the one element of matches
   else
      throw a ReferenceError exception — this access is ambiguous because it found several different instance properties
            in the same class
   end if
end proc;
```

findArchetypeProperty(*o*, *multiname*, *access*, *flat*) looks in object *o* for any local or inherited property with one of the names in *multiname* and access that includes *access*. If *flat* is **true**, then properties inherited from the archetype are not considered in the search. If it finds no property, *findArchetypeProperty* returns **none**. If it finds one property, *findArchetypeProperty* returns it. If it finds more than one property, *findArchetypeProperty* prefers the more local one in the list of *o*'s superclasses or archetypes; if two or more properties remain, the singleton one is preferred; if two or more properties still remain, *findArchetypeProperty* throws an error.

Note that *findArchetypeProperty*(*o*, *multiname*, *access*, *flat*) searches *o* itself rather than *o*'s class for properties. *findArchetypeProperty* will not find instance properties unless *o* is a class.

```
proc findArchetypeProperty(o: OBJECT, multiname: MULTINAME, access: ACCESS, flat: BOOLEAN): PROPERTYOPT
    m: PROPERTYOPT;
    case o of
        UNDEFINED □ NULL □ BOOLEAN □ LONG □ ULONG □ FLOAT32 □ FLOAT64 □ CHAR16 □ STRING □
                NAMESPACE □ COMPOUNDATTRIBUTE □ METHODCLOSURE do
            m □ none;
        SIMPLEINSTANCE □ REGEXP □ DATE □ PACKAGE do
            m □ findLocalSingletonProperty(o, multiname, access);
        CLASS do m □ findClassProperty(o, multiname, access)
    end case;
    if m ≠ none then return m end if;
    if flat then return none end if;
    a: OBJECTOPT □ archetype(o);
    if a = none then return none end if;
    return findArchetypeProperty(a, multiname, access, flat)
end proc;
```

```
proc findClassProperty(c: CLASS, multiname: MULTINAME, access: ACCESS): PROPERTYOPT
    m: PROPERTYOPT □ findLocalSingletonProperty(c, multiname, access);
    if m = none then
        m □ findLocalInstanceProperty(c, multiname, access);
        if m = none then
            super: CLASSOPT □ c.super;
            if super ≠ none then m □ findClassProperty(super, multiname, access) end if
        end if
    end if;
    return m
end proc;
```

findBaseInstanceProperty(*c*, *multiname*, *accesses*) looks in class *c* and its ancestors for an instance property with one of the names in *multiname* and accesses that have a nonempty intersection with *accesses*. If there is no such property, *findBaseInstanceProperty* returns **none**. If there is exactly one such property, *findBaseInstanceProperty* returns it. If there is more than one such property, *findBaseInstanceProperty* prefers the one defined in the least specific class; if two or more properties still remain, *findBaseInstanceProperty* throws an error.

```
proc findBaseInstanceProperty(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
    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: INSTANCEPROPERTYOPT □ findLocalInstanceProperty(s, multiname, accesses);
        if m ≠ none then return m end if
    end for each;
    return none
end proc;
```

getDerivedInstanceProperty(*c*, *mBase*, *accesses*) returns the most derived instance property whose name includes that of *mBase* and whose accesses that have a nonempty intersection with *accesses*. The caller of *getDerivedInstanceProperty* ensures that such an instance property 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 getDerivedInstanceProperty(c: CLASS, mBase: INSTANCEPROPERTY, accesses: ACCESSSET): INSTANCEPROPERTY
   if some m □ c.instanceProperties satisfies mBase.multiname □ m.multiname and
         accessesOverlap(instancePropertyAccesses(m), accesses) then
      return m
   else return getDerivedInstanceProperty(c.super, mBase, accesses)
   end if
end proc;
```

readImplicitThis(*env*) returns the value of implicit `this` to be used to access instance properties within a class's scope without using the `.` operator. An implicit `this` is well-defined only inside instance methods and constructors; *readImplicitThis* throws an error if there is no well-defined implicit `this` value or if an attempt is made to read it before it has been initialised.

```
proc readImplicitThis(env: ENVIRONMENT): OBJECT
   frame: PARAMETERFRAMEOPT □ getEnclosingParameterFrame(env);
   if frame = none then
      throw a ReferenceError exception — can't access instance properties outside an instance method without supplying
            an instance object
   end if;
   this: OBJECTOPT □ frame.this;
   if this = none then
      throw a ReferenceError exception — can't access instance properties inside a non-instance method without
            supplying an instance object
   end if;
   if frame.kind □ {instanceFunction, constructorFunction} then
      throw a ReferenceError exception — can't access instance properties inside a non-instance method without
            supplying an instance object
   end if;
   if not frame.superconstructorCalled then
      throw an UninitializedError exception — can't access instance properties from within a constructor before the
            superconstructor has been called
   end if;
   return this
end proc;
```

hasProperty(*o*, *property*, *flat*, *phase*) returns **true** if *o* has a readable or writable property named *property*. If *flat* is **true**, then properties inherited from the archetype are not considered.

```
proc hasProperty(o: OBJECT, property: OBJECT, flat: BOOLEAN, phase: PHASE): BOOLEAN
   c: CLASS □ objectType(o);
   return c.hasProperty(o, c, property, flat, phase)
end proc;
```

hasProperty(*o*, *c*, *property*, *flat*, *phase*) is the implementation of *hasProperty* for a native class unless specified otherwise in the class's definition. Host classes may either also use *ordinaryHasProperty* or define a different procedure to perform this test. *c* is *o*'s type.

```
proc ordinaryHasProperty(o: OBJECT, c: CLASS, property: OBJECT, flat: BOOLEAN, phase: PHASE): BOOLEAN
   qname: QUALIFIEDNAME □ objectToQualifiedName(property, phase);
   return findBaseInstanceProperty(c, {qname}, read) ≠ none or
         findBaseInstanceProperty(c, {qname}, write) ≠ none or
         findArchetypeProperty(o, {qname}, read, flat) ≠ none or
         findArchetypeProperty(o, {qname}, write, flat) ≠ none
end proc;
```

10.7 Reading

If *r* is an OBJECT, *readReference*(*r*, *phase*) returns it unchanged. If *r* is a REFERENCE, *readReference* reads *r* and returns the result. If *phase* is **compile**, only constant 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.multiname, none, true, phase);
      BRACKETREFERENCE do
         result ← r.limit.bracketRead(r.base, r.limit, r.args, true, phase)
   end case;
   if result ≠ none then return result
   else
      throw a ReferenceError exception — property not found, and no default value is available
   end if
end proc;
```

dotRead(*o*, *multiname*, *phase*) reads and returns the value of the *multiname* property of *o*. *dotRead* throws an error if the property does not exist and no default value was available for it.

```
proc dotRead(o: OBJECT, multiname: MULTINAME, phase: PHASE): OBJECT
   limit: CLASS ← objectType(o);
   result: OBJECTOPT ← limit.read(o, limit, multiname, none, true, phase);
   if result = none then
      throw a ReferenceError exception — property not found, and no default value is available
   end if;
   return result
end proc;
```

readLength(*o*, *phase*) reads and returns the value of the `length` property of *o*, ensuring that it is an integer between 0 and *arrayLimit* inclusive.

```
proc readLength(o: OBJECT, phase: PHASE): INTEGER
   value: OBJECT ← dotRead(o, {public::“length”}, phase);
   if value ∉ GENERALNUMBER then throw a TypeError exception — length not an integer
   end if;
   length: INTEGEROPT ← checkInteger(value);
   if length = none then throw a RangeError exception — length not an integer
   elsif 0 ≤ length ≤ arrayLimit then return length
   else throw a RangeError exception — length out of range
   end if
end proc;
```

indexRead(*o*, *i*, *phase*) returns the value of `o[i]` or **none** if no such property was found; unlike *dotRead*, *indexRead* does not return a default value for missing properties. *i* should always be a valid array index.

```
proc indexRead(o: OBJECT, i: INTEGER, phase: PHASE): OBJECTOPT
   note 0 ≤ i < arrayLimit;
   limit: CLASS ← objectType(o);
   x: FLOAT64 ← $i_{f64}$;
   result: OBJECTOPT ← limit.bracketRead(o, limit, [x], false, phase);
   if result ≠ none and not hasProperty(o, x, true, phase) then
      At the implementation’s discretion either do nothing, set result to none, or throw a ReferenceError.
   end if;
   return result
end proc;
```

ordinaryBracketRead(*o*, *limit*, *args*, *undefinedIfMissing*, *phase*) evaluates the expression `o[args]` when *o* is a native object. Host objects may either also use *ordinaryBracketRead* or choose a different procedure *P* to evaluate `o[args]` by writing *P* into *objectType*(*o*).bracketRead.

limit is used to handle the expression `super(o)[args]`, in which case *limit* is the superclass of the class inside which the `super` expression appears. Otherwise, *limit* is set to *objectType*(*o*).

```
proc ordinaryBracketRead(o: OBJECT, limit: CLASS, args: OBJECT[], undefinedIfMissing: BOOLEAN, phase: PHASE):
        OBJECTOPT
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    qname: QUALIFIEDNAME ← objectToQualifiedName(args[0], phase);
    return limit.read(o, limit, {qname}, none, undefinedIfMissing, phase)
end proc;

proc lexicalRead(env: ENVIRONMENT, multiname: MULTINAME, phase: PHASE): OBJECT
    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, env, false, phase);
            PARAMETERFRAME ∪ LOCALFRAME do
                m: SINGLETONPROPERTYOPT ← findLocalSingletonProperty(frame, multiname, read);
                if m ≠ none then result ← readSingletonProperty(m, phase) end if;
            WITHFRAME do
                value: OBJECTOPT ← frame.value;
                if value = none then
                    case phase of
                        {compile} do
                            throw a ConstantError exception — cannot read a with statement’s frame from a constant
                                    expression;
                        {run} do
                            throw an UninitializedError exception — cannot read a with statement’s frame before that
                                    statement’s expression has been evaluated
                    end case
                end if;
                limit: CLASS ← objectType(value);
                result ← limit.read(value, limit, multiname, env, false, phase)
        end case;
        if result ≠ none then return result end if;
        i ← i + 1
    end while;
    throw a ReferenceError exception — no property found with the name multiname
end proc;
```

```
proc ordinaryRead(o: Object, limit: Class, multiname: Multiname, env: EnvironmentOpt,
      undefinedIfMissing: Boolean, phase: Phase): ObjectOpt
   mBase: InstancePropertyOpt ← findBaseInstanceProperty(limit, multiname, read);
   if mBase ≠ none then return readInstanceProperty(o, limit, mBase, phase) end if;
   if limit ≠ objectType(o) then return none end if;
   flat: Boolean ← env ≠ none and o ∈ Class;
   m: PropertyOpt ← findArchetypeProperty(o, multiname, read, flat);
   case m of
      {none} do
         if undefinedIfMissing and o ∈ SimpleInstance ∪ Date ∪ RegExp ∪ Package and not o.sealed then
            case phase of
               {compile} do
                  throw a ConstantError exception — a constant expression cannot read dynamic properties;
               {run} do return undefined
            end case
         else return none
         end if;
      SingletonProperty do return readSingletonProperty(m, phase);
      InstanceProperty do
         if o ∈ Class or env = none then
            throw a ReferenceError exception — cannot read an instance property without supplying an instance
         end if;
         this: Object ← readImplicitThis(env);
         return readInstanceProperty(this, objectType(this), m, phase)
   end case
end proc;
```

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

```
proc readInstanceSlot(o: Object, qname: QualifiedName, phase: Phase): Object
   c: Class ← objectType(o);
   mBase: InstancePropertyOpt ← findBaseInstanceProperty(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 readInstanceProperty(o, c, mBase, phase)
end proc;
```

```
proc readInstanceProperty(this: OBJECT, c: CLASS, mBase: INSTANCEPROPERTY, phase: PHASE): OBJECT
   m: INSTANCEPROPERTY ← getDerivedInstanceProperty(c, mBase, read);
   case m of
      INSTANCEVARIABLE do
         if phase = compile and not m.immutable then
            throw a ConstantError exception — a constant expression cannot read mutable variables
         end if;
         v: OBJECTOPT ← findSlot(this, m).value;
         if v = none then
            case phase of
               {compile} do
                  throw a ConstantError exception — cannot read uninitalised const variables from a constant
                        expression;
               {run} do
                  throw an UninitializedError exception — cannot read a const instance variable before it is initialised
            end case
         end if;
         return v;
      INSTANCEMETHOD do
         slots: SLOT{} ← {new SLOT⟨⟨id: ivarFunctionLength, value: (m.length)f64⟩⟩};
         return METHODCLOSURE⟨⟨this: this, method: m, slots: slots⟩⟩
      INSTANCEGETTER do return m.call(this, phase);
      INSTANCESETTER do
         m cannot be an INSTANCESETTER because these are only represented as write-only properties.
   end case
end proc;
```

```
proc readSingletonProperty(m: SINGLETONPROPERTY, phase: PHASE): OBJECT
  case m of
    {forbidden} do
      throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if
          any block inside the current region shadows it;
    DYNAMICVAR do
      if phase = compile then
        throw a ConstantError exception — a constant expression cannot read mutable variables
      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 a ConstantError exception — a constant expression cannot read mutable variables
      end if;
      value: VARIABLEVALUE ← m.value;
      case value of
        OBJECT do return value;
        {none} do
          if not m.immutable then throw an UninitializedError exception end if;
          note  Try to run a const variable's initialiser if there is one.
          Evaluate setupVariable(m) and ignore its result;
          initializer: INITIALIZER ∪ {none, busy} ← m.initializer;
          if initializer ∈ {none, busy} then
            case phase of
              {compile} do
                throw a ConstantError exception — a constant expression cannot access a constant with a
                    missing or recursive initialiser;
              {run} do throw an UninitializedError exception
            end case
          end if;
          m.initializer ← busy;
          coercedValue: OBJECT;
          try
            newValue: OBJECT ← initializer(m.initializerEnv, compile);
            coercedValue ← writeVariable(m, newValue, true)
          catch x: SEMANTICEXCEPTION do
            note  If initialisation failed, restore m.initializer to its original value so it can be tried later.
            m.initializer ← initializer;
            throw x
          end try;
          return coercedValue;
        UNINSTANTIATEDFUNCTION do
          note  An uninstantiated function can only be found when phase = compile.
          throw a ConstantError exception — an uninstantiated function is not a constant expression
      end case;
    GETTER do
      env: ENVIRONMENTOPT ← m.env;
      if env = none then
        note  An uninstantiated getter can only be found when phase = compile.
        throw a ConstantError exception — an uninstantiated getter is not a constant expression
      end if;
      return m.call(env, phase);
    SETTER do
      m cannot be a SETTER because these are only represented as write-only properties.
  end case
```

```
   end proc;
```

10.8 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 constant expression.

```
   proc writeReference(r: OBJORREF, newValue: OBJECT, phase: {run})
      result: {none, ok};
      case r of
         OBJECT do
            throw a ReferenceError exception — a non-reference is not a valid target of an assignment;
         LEXICALREFERENCE do
            Evaluate lexicalWrite(r.env, r.variableMultiname, newValue, not r.strict, phase) and ignore its result;
            result ← ok;
         DOTREFERENCE do
            result ← r.limit.write(r.base, r.limit, r.multiname, none, newValue, true, phase);
         BRACKETREFERENCE do
            result ← r.limit.bracketWrite(r.base, r.limit, r.args, newValue, true, phase)
      end case;
      if result = none then
         throw a ReferenceError exception — property not found and could not be created
      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, none, newValue, true, phase);
      if result = none then
         throw a ReferenceError exception — property not found and could not be created
      end if
   end proc;
```

writeLength(*o*, *length*, *phase*) ensures that *length* is between 0 and *arrayLimit* inclusive and then writes it into the `length` property of *o*. Note that if *o* is an `Array`, the act of writing its `length` property will invoke the *Array_setLength* setter.

```
   proc writeLength(o: OBJECT, length: INTEGER, phase: {run})
      if length < 0 or length > arrayLimit then
         throw a RangeError exception — length out of range
      end if;
      Evaluate dotWrite(o, {public::"length"}, $length_{f64}$, phase) and ignore its result
   end proc;
```

```
proc indexWrite(o: OBJECT, i: INTEGER, newValue: OBJECTOPT, phase: {run})
   if $i < 0$ or $i \geq arrayLimit$ then throw a RangeError exception — index out of range
   end if;
   limit: CLASS  objectType(o);
   if newValue = none then
      deleteResult: BOOLEANOPT  limit.bracketDelete(o, limit, [$i_{f64}$], phase);
      if deleteResult = false then
         throw a ReferenceError exception — cannot delete element
      end if
   else
      writeResult: {none, ok}  limit.bracketWrite(o, limit, [$i_{f64}$], newValue, true, phase);
      if writeResult = none then
         throw a ReferenceError exception — element not found and could not be created
      end if
   end if
end proc;

proc ordinaryBracketWrite(o: OBJECT, limit: CLASS, args: OBJECT[], newValue: OBJECT, createIfMissing: BOOLEAN,
      phase: {run}): {none, ok}
   if $|args| \neq 1$ then
      throw an ArgumentError exception — exactly one argument must be supplied
   end if;
   qname: QUALIFIEDNAME  objectToQualifiedName(args[0], phase);
   return limit.write(o, limit, {qname}, none, newValue, createIfMissing, phase)
end proc;
```

```
proc lexicalWrite(env: ENVIRONMENT, multiname: MULTINAME, newValue: OBJECT, createIfMissing: BOOLEAN,
        phase: {run})
    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, env, newValue, false, phase);
            PARAMETERFRAME ∪ LOCALFRAME do
                m: SINGLETONPROPERTYOPT ← findLocalSingletonProperty(frame, multiname, write);
                if $m \neq$ none then
                    Evaluate writeSingletonProperty(m, newValue, phase) and ignore its result;
                    result ← ok
                end if;
            WITHFRAME do
                value: OBJECTOPT ← frame.value;
                if value = none then
                    throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s
                            expression has been evaluated
                end if;
                limit: CLASS ← objectType(value);
                result ← limit.write(value, limit, multiname, env, newValue, false, 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, env, newValue, true, phase);
        if result = ok then return end if
    end if;
    throw a ReferenceError exception — no existing property found with the name multiname and one could not be
            created
end proc;
```

```
proc ordinaryWrite(o: Object, limit: Class, multiname: Multiname, env: EnvironmentOpt, newValue: Object,
        createIfMissing: Boolean, phase: {run}): {none, ok}
    mBase: InstancePropertyOpt ← findBaseInstanceProperty(limit, multiname, write);
    if mBase ≠ none then
        Evaluate writeInstanceProperty(o, limit, mBase, newValue, phase) and ignore its result;
        return ok
    end if;
    if limit ≠ objectType(o) then return none end if;
    m: PropertyOpt ← findArchetypeProperty(o, multiname, write, true);
    case m of
        {none} do
            if createIfMissing and o ∈ SimpleInstance ∪ Date ∪ RegExp ∪ Package and not o.sealed and
                    (some qname ∈ multiname satisfies qname.namespace = public) then
                note  Before trying to create a new dynamic property named qname, check that there is no read-only fixed
                      property with the same name.
                if findBaseInstanceProperty(objectType(o), {qname}, read) = none and
                        findArchetypeProperty(o, {qname}, read, true) = none then
                    Evaluate createDynamicProperty(o, qname, false, true, newValue) and ignore its result;
                    return ok
                end if
            end if;
            return none;
        SingletonProperty do
            Evaluate writeSingletonProperty(m, newValue, phase) and ignore its result;
            return ok;
        InstanceProperty do
            if o ∈ Class or env = none then
                throw a ReferenceError exception — cannot write an instance property without supplying an instance
            end if;
            this: Object ← readImplicitThis(env);
            Evaluate writeInstanceProperty(this, objectType(this), m, newValue, phase) and ignore its result;
            return ok
    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 ∪ {LocalBinding⟨⟨qname: qname, accesses: readWrite, explicit: false,
            enumerable: enumerable, content: dv⟩⟩}
end proc;
```

```
proc writeInstanceProperty(this: OBJECT, c: CLASS, mBase: INSTANCEPROPERTY, newValue: OBJECT, phase: {run})
   m: INSTANCEPROPERTY ← getDerivedInstanceProperty(c, mBase, write);
   case m of
      INSTANCEVARIABLE do
         s: SLOT ← findSlot(this, m);
         coercedValue: OBJECT ← coerce(newValue, m.type);
         if m.immutable and s.value ≠ none then
            throw a ReferenceError exception — cannot initialise a const instance variable twice
         end if;
         s.value ← coercedValue;
      INSTANCEMETHOD do
         throw a ReferenceError exception — cannot write to an instance method;
      INSTANCEGETTER do
         m cannot be an INSTANCEGETTER because these are only represented as read-only properties.
      INSTANCESETTER do Evaluate m.call(this, newValue, phase) and ignore its result
   end case
end proc;
```

```
proc writeSingletonProperty(m: SINGLETONPROPERTY, newValue: OBJECT, phase: {run})
   case m of
      {forbidden} do
         throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if
               any block inside the current region shadows it;
      VARIABLE do Evaluate writeVariable(m, newValue, false) and ignore its result;
      DYNAMICVAR do m.value ← newValue;
      GETTER do
         m cannot be a GETTER because these are only represented as read-only properties.
      SETTER do
         env: ENVIRONMENTOPT ← m.env;
         note  All instances are resolved for the run phase, so env ≠ none.
         Evaluate m.call(newValue, env, phase) and ignore its result
   end case
end proc;
```

10.9 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 constant expression.

```
proc deleteReference(r: OBJORREF, strict: BOOLEAN, phase: {run}): BOOLEAN
   result: BOOLEANOPT;
   case r of
      OBJECT do
         if strict then
            throw a ReferenceError exception — a non-reference is not a valid target for delete in strict mode
         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.multiname, none, 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 ordinaryBracketDelete(o: OBJECT, limit: CLASS, args: OBJECT[], phase: {run}): BOOLEANOPT
   if |args| ≠ 1 then
      throw an ArgumentError exception — exactly one argument must be supplied
   end if;
   qname: QUALIFIEDNAME ← objectToQualifiedName(args[0], phase);
   return limit.delete(o, limit, {qname}, none, phase)
end proc;

proc lexicalDelete(env: ENVIRONMENT, multiname: MULTINAME, phase: {run}): BOOLEAN
   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, env, phase);
         PARAMETERFRAME ∪ LOCALFRAME do
            if findLocalSingletonProperty(frame, multiname, write) ≠ none then
               result ← false
            end if;
         WITHFRAME do
            value: OBJECTOPT ← frame.value;
            if value = none then
               throw an UninitializedError exception — cannot read a with statement's frame before that statement's
                     expression has been evaluated
            end if;
            limit: CLASS ← objectType(value);
            result ← limit.delete(value, limit, multiname, env, phase)
      end case;
      if result ≠ none then return result end if;
      i ← i + 1
   end while;
   return true
end proc;
```

```
proc ordinaryDelete(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, phase: {run}):
        BOOLEANOPT
    if findBaseInstanceProperty(limit, multiname, write) ≠ none then return false end if;
    if limit ≠ objectType(o) then return none end if;
    m: PROPERTYOPT ← findArchetypeProperty(o, multiname, write, true);
    case m of
        {none} do return none;
        {forbidden} do
            throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if
                    any block inside the current region shadows it;
        VARIABLE ∪ 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;
        INSTANCEPROPERTY do
            if o ∈ CLASS or env = none then return false end if;
            Evaluate readImplicitThis(env) and ignore its result;
            return false
    end case
end proc;
```

10.10 Enumerating

```
proc ordinaryEnumerate(o: OBJECT): OBJECT{}
    e1: OBJECT{} ← enumerateInstanceProperties(objectType(o));
    e2: OBJECT{} ← enumerateArchetypeProperties(o);
    return e1 ∪ e2
end proc;
```

```
proc enumerateInstanceProperties(c: CLASS): OBJECT{}
    e: OBJECT{} ← {};
    for each m ∈ c.instanceProperties 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 ∪ enumerateInstanceProperties(super)
    end if
end proc;
```

```
proc enumerateArchetypeProperties(o: OBJECT): OBJECT{}
    e: OBJECT{} ← {};
    for each a ∈ {o} ⊕ archetypes(o) do
        if a ∈ BINDINGOBJECT then e ← e ∪ enumerateSingletonProperties(a) end if
    end for each;
    return e
end proc;
```

```
proc enumerateSingletonProperties(o: BindingObject): Object{}
   e: Object{} ← {};
   for each b $\in$ o.localBindings do
      if b.enumerable and b.qname.namespace = public then e ← e $\cup$ {b.qname.id} end if
   end for each;
   if o $\in$ Class then
      super: ClassOpt ← o.super;
      if super $\neq$ none then e ← e $\cup$ enumerateSingletonProperties(super) end if
   end if;
   return e
end proc;
```

10.11 Calling Instances

```
proc call(this: Object, a: Object, args: Object[], phase: Phase): Object
   case a of
      Undefined $\cup$ Null $\cup$ Boolean $\cup$ GeneralNumber $\cup$ Char16 $\cup$ String $\cup$ Namespace $\cup$
            CompoundAttribute $\cup$ Date $\cup$ RegExp $\cup$ Package do
         throw a TypeError exception;
      Class do return a.call(this, a, args, phase);
      SimpleInstance do
         f: (Object $\times$ SimpleInstance $\times$ Object[] $\times$ Phase $\to$ Object) $\cup$ {none} ← a.call;
         if f = none then throw a TypeError exception end if;
         return f(this, a, args, phase);
      MethodClosure do
         m: InstanceMethod ← a.method;
         return m.call(a.this, args, phase)
   end case
end proc;

proc ordinaryCall(this: Object, c: Class, args: Object[], phase: Phase): Object
   note  This function can be used in a constant expression.
   if not c.complete then
      throw a ConstantError exception — cannot call a class before its definition has been compiled
   end if;
   if |args| $\neq$ 1 then
      throw an ArgumentError exception — exactly one argument must be supplied
   end if;
   return coerce(args[0], c)
end proc;

proc sameAsConstruct(this: Object, c: Class, args: Object[], phase: Phase): Object
   return construct(c, args, phase)
end proc;
```

10.12 Creating Instances

```
proc construct(a: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
   case a of
      UNDEFINED □ NULL □ BOOLEAN □ GENERALNUMBER □ CHAR16 □ STRING □ NAMESPACE □
            COMPOUNDATTRIBUTE □ METHODCLOSURE □ DATE □ REGEXP □ PACKAGE do
         throw a TypeError exception;
      CLASS do return a.construct(a, args, phase);
      SIMPLEINSTANCE do
         f: (SIMPLEINSTANCE □ OBJECT[] □ PHASE □  OBJECT) □ {none} □  a.construct;
         if f = none then throw a TypeError exception end if;
         return f(a, args, phase)
   end case
end proc;

proc ordinaryConstruct(c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
   if not c.complete then
      throw a ConstantError exception — cannot construct an instance of a class before its definition has been compiled
   end if;
   if phase = compile then
      throw a ConstantError exception — a class constructor call is not a constant expression because it evaluates to a
            new object each time it is evaluated
   end if;
   this: SIMPLEINSTANCE □  createSimpleInstance(c, c.prototype, none, none, none);
   Evaluate callInit(this, c, args, phase) and ignore its result;
   return this
end proc;

proc createSimpleInstance(c: CLASS, archetype: 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.instanceProperties 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: {}, archetype: archetype, sealed: not c.dynamic, type: c, slots: slots,
         call: call, construct: construct, env: env□
end proc;

proc callInit(this: SIMPLEINSTANCE, c: CLASSOPT, args: OBJECT[], phase: {run})
   init: (SIMPLEINSTANCE □ OBJECT[] □ {run} □  ()) □ {none} □  none;
   if c ≠ none then init □  c.init end if;
   if init ≠ none then Evaluate init(this, args, phase) and ignore its result
   else
      if args ≠ [] then
         throw an ArgumentError exception — the default constructor does not take any arguments
      end if
   end if
end proc;
```

10.13 Adding Local Definitions

```
proc defineSingletonProperty(env: Environment, id: String, namespaces: Namespace{},
      overrideMod: OverrideModifier, explicit: Boolean, accesses: AccessSet, m: SingletonProperty):
      Multiname
   innerFrame: NonWithFrame ← env[0];
   if overrideMod ≠ none then
      throw an AttributeError exception — a local definition cannot have the override attribute
   end if;
   if explicit and innerFrame ∉ Package then
      throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
   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 a DefinitionError exception — duplicate definition in the same scope
   end if;
   if innerFrame ∈ Class and id = innerFrame.name then
      throw a DefinitionError exception — a static property of a class cannot have the same name as the class,
            regardless of the namespace
   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 a DefinitionError exception — this definition would shadow a property defined in an outer scope within
               the same region
      end if
   end for each;
   newBindings: LocalBinding{} ← {LocalBinding⟨qname: qname, accesses: accesses, explicit: explicit,
         enumerable: true, content: m⟩ | ∀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, explicit: true,
         enumerable: true, content: forbidden⟩ | ∀qname ∈ multiname};
   for each frame ∈ regionalEnv[1 ...] do
      note Since frame ∉ Class here, a Class frame never gets a forbidden binding.
      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, explicit: false, enumerable: true, content: v⟩⟩};
      return v
   elsif |existingBindings| ≠ 1 then
      throw a DefinitionError exception — a hoisted definition conflicts with a non-hoisted one
   else
      b: LOCALBINDING ← the one element of existingBindings;
      m: SINGLETONPROPERTY ← b.content;
      if b.accesses ≠ readWrite or m ∉ DYNAMICVAR then
         throw a DefinitionError exception — a hoisted definition conflicts with a non-hoisted one
      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;
```

10.14 Adding Instance Definitions

```
proc searchForOverrides(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
   mBase: INSTANCEPROPERTYOPT ← none;
   s: CLASSOPT ← c.super;
   if s ≠ none then
      for each qname ∈ multiname do
         m: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(s, {qname}, accesses);
         if mBase = none then mBase ← m
         elsif m ≠ none and m ≠ mBase then
            throw a DefinitionError exception — cannot override two separate superclass methods at the same time
         end if
      end for each
   end if;
   return mBase
end proc;
```

```
proc defineInstanceProperty(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE{},
      overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, m: INSTANCEPROPERTY): INSTANCEPROPERTYOPT
   if explicit then
      throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
   end if;
   accesses: ACCESSSET ← instancePropertyAccesses(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: INSTANCEPROPERTYOPT ← searchForOverrides(c, searchedMultiname, accesses);
   mOverridden: INSTANCEPROPERTYOPT ← none;
   if mBase ≠ none then
      mOverridden ← getDerivedInstanceProperty(c, mBase, accesses);
      definedMultiname ← mOverridden.multiname;
      if not (requestedMultiname ⊆ definedMultiname) then
         throw a DefinitionError exception — cannot extend the set of a property's namespaces when overriding it
      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 not goodKind then
         throw a DefinitionError exception — a method can override only another method, a variable can override only
               another variable, a getter can override only a getter or a variable, and a setter can override only a setter or a
               variable
      end if;
      if mOverridden.final then
         throw a DefinitionError exception — cannot override a final property
      end if
   end if;
   if some m2 ∈ c.instanceProperties satisfies m2.multiname ∩ definedMultiname ≠ {} and
         accessesOverlap(instancePropertyAccesses(m2), accesses) then
      throw a DefinitionError exception — duplicate definition in the same scope
   end if;
   case overrideMod of
      {none} do
         if mBase ≠ none then
            throw a DefinitionError exception — a definition that overrides a superclass's property must be marked with
                  the override attribute
         end if;
         if searchForOverrides(c, openMultiname, accesses) ≠ none then
            throw a DefinitionError exception — this definition is hidden by one in a superclass when accessed without a
                  namespace qualifier; in the rare cases where this is intentional, use the override(false) attribute
         end if;
      {false} do
```

```
            if mBase ≠ none then
               throw a DefinitionError exception — this definition is marked with override(false) but it overrides a
                     superclass's property
            end if;
         {true} do
            if mBase = none then
               throw a DefinitionError exception — this definition is marked with override or override(true) but it
                     doesn't override a superclass's property
            end if;
         {undefined} do nothing
      end case;
      m.multiname ← definedMultiname;
      c.instanceProperties ← c.instanceProperties ∪ {m};
      return mOverridden
   end proc;
```

10.15 Instantiation

```
   proc instantiateFunction(uf: UNINSTANTIATEDFUNCTION, env: ENVIRONMENT): SIMPLEINSTANCE
      c: CLASS ← uf.type;
      i: SIMPLEINSTANCE ← createSimpleInstance(c, c.prototype, uf.call, uf.construct, env);
      Evaluate dotWrite(i, {public::"length"}, (uf.length)f64, run) and ignore its result;
      if c = PrototypeFunction then
         prototype: OBJECT ← construct(Object, [], run);
         Evaluate dotWrite(prototype, {public::"constructor"}, i, run) and ignore its result;
         Evaluate dotWrite(i, {public::"prototype"}, prototype, run) and ignore its result
      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 instantiateProperty(m: SingletonProperty, env: Environment): SingletonProperty
   case m of
      {forbidden} 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,
               initializer: m.initializer, initializerEnv: 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;
            {none} do return new Getter□call: m.call, env: env□
         end case;
      Setter do
         case m.env of
            Environment do return m;
            {none} do return new Setter□call: m.call, env: env□
         end case
   end case
end proc;

tuple PropertyTranslation
   from: SingletonProperty,
   to: SingletonProperty
end tuple;

proc instantiateLocalFrame(frame: LocalFrame, env: Environment): LocalFrame
   instantiatedFrame: LocalFrame □  new LocalFrame□localBindings: {}□
   properties: SingletonProperty{} □  {b.content | □b □ frame.localBindings};
   propertyTranslations: PropertyTranslation{} □  {PropertyTranslation□from: m,
         to: instantiateProperty(m, [instantiatedFrame] ⊕ env)□| □m □ properties};
   proc translateProperty(m: SingletonProperty): SingletonProperty
      mi: PropertyTranslation □  the one element mi □ propertyTranslations that satisfies mi.from = m;
      return mi.to
   end proc;
   instantiatedFrame.localBindings □  {LocalBinding□content: translateProperty(b.content), other fields from b□|
         □b □ frame.localBindings};
   return instantiatedFrame
end proc;
```

```
proc instantiateParameterFrame(frame: PARAMETERFRAME, env: ENVIRONMENT, singularThis: OBJECTOPT):
      PARAMETERFRAME
   note  frame.superconstructorCalled must be true if and only if frame.kind is not constructorFunction.
   instantiatedFrame: PARAMETERFRAME □  new PARAMETERFRAME□localBindings: {}, kind: frame.kind,
         handling: frame.handling, callsSuperconstructor: frame.callsSuperconstructor,
         superconstructorCalled: frame.superconstructorCalled, this: singularThis, returnType: frame.returnType□;
   note  properties will contain the set of all SINGLETONPROPERTY records found in the frame.
   properties: SINGLETONPROPERTY{} □  {b.content | □b □ frame.localBindings};
   note  If any of the parameters (including the rest parameter) are anonymous, their bindings will not be present in
         frame.localBindings. In this situation, the following steps add their SINGLETONPROPERTY records to properties.
   for each p □ frame.parameters do properties □  properties □  {p.var} end for each;
   rest: VARIABLEOPT □  frame.rest;
   if rest ≠ none then properties □  properties □  {rest} end if;
   propertyTranslations: PROPERTYTRANSLATION{} □  {PROPERTYTRANSLATION□from: m,
         to: instantiateProperty(m, [instantiatedFrame] ⊕ env)□| □m □ properties};
   proc translateProperty(m: SINGLETONPROPERTY): SINGLETONPROPERTY
      mi: PROPERTYTRANSLATION □  the one element mi □ propertyTranslations that satisfies mi.from = m;
      return mi.to
   end proc;
   instantiatedFrame.localBindings □  {LOCALBINDING□content: translateProperty(b.content), other fields from b□|
         □b □ frame.localBindings};
   instantiatedFrame.parameters □  [PARAMETER□var: translateProperty(op.var), default: op.default□|
         □op □ frame.parameters];
   if rest = none then instantiatedFrame.rest □  none
   else instantiatedFrame.rest □  translateProperty(rest)
   end if;
   return instantiatedFrame
end proc;
```

10.16 Sealing

```
proc sealObject(o: OBJECT)
   if o □ SIMPLEINSTANCE □ REGEXP □ DATE □ PACKAGE then o.sealed □  true end if
end proc;

proc sealAllLocalProperties(o: OBJECT)
   if o □ BINDINGOBJECT then
      for each b □ o.localBindings do
         m: SINGLETONPROPERTY □  b.content;
         if m □ DYNAMICVAR then m.sealed □  true end if
      end for each
   end if
end proc;

proc sealLocalProperty(o: OBJECT, qname: QUALIFIEDNAME)
   c: CLASS □  objectType(o);
   if findBaseInstanceProperty(c, {qname}, read) = none and
         findBaseInstanceProperty(c, {qname}, write) = none and o □ BINDINGOBJECT then
      matchingProperties: SINGLETONPROPERTY{} □  {b.content | □b □ o.localBindings such that b.qname = qname};
      for each m □ matchingProperties do
         if m □ DYNAMICVAR then m.sealed □  true end if
      end for each
   end if
end proc;
```

10.17 Standard Class Utilities

```
proc defaultArg(args: OBJECT[], n: INTEGER, default: OBJECT): OBJECT
   if n ≥ |args| then return default end if;
   arg: OBJECT ← args[n];
   if arg = undefined then return default else return arg end if
end proc;

proc stdConstBinding(qname: QUALIFIEDNAME, type: CLASS, value: OBJECT): LOCALBINDING
   return LOCALBINDING⟨qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
         new VARIABLE⟨type: type, value: value, immutable: true, setup: none, initializer: none⟩⟩
end proc;

proc stdExplicitConstBinding(qname: QUALIFIEDNAME, type: CLASS, value: OBJECT): LOCALBINDING
   return LOCALBINDING⟨qname: qname, accesses: readWrite, explicit: true, enumerable: false, content:
         new VARIABLE⟨type: type, value: value, immutable: true, setup: none, initializer: none⟩⟩
end proc;

proc stdVarBinding(qname: QUALIFIEDNAME, type: CLASS, value: OBJECT): LOCALBINDING
   return LOCALBINDING⟨qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
         new VARIABLE⟨type: type, value: value, immutable: false, setup: none, initializer: none⟩⟩
end proc;

proc stdFunction(qname: QUALIFIEDNAME, call: OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT,
      length: INTEGER): LOCALBINDING
   slots: SLOT{} ← {new SLOT⟨id: ivarFunctionLength, value: length_f64⟩};
   f: SIMPLEINSTANCE ← new SIMPLEINSTANCE⟨localBindings: {}, archetype: FunctionPrototype, sealed: true,
         type: Function, slots: slots, call: call, construct: none, env: none⟩
   return LOCALBINDING⟨qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
         new VARIABLE⟨type: Function, value: f, immutable: true, setup: none, initializer: none⟩⟩
end proc;
```

stdReserve(*qname*, *archetype*) is used during the creation of system objects. It returns an alias of the local binding of *qname* in *archetype*, which should be the archetype of the object being created. The alias that *stdReserve* defines serves to prevent *qname* from being later redefined by users in the object being created while at the same time retaining the definition of *qname* that would normally be inherited from *archetype*.

```
proc stdReserve(qname: QUALIFIEDNAME, archetype: SIMPLEINSTANCE): LOCALBINDING
   matchingBindings: LOCALBINDING{} ← {b | ∀b ∈ archetype.localBindings such that b.qname = qname};
   return the one element of matchingBindings
end proc;
```

~~12~~11 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).

11.1 Terminal Actions

```
Name[Identifier]: STRING;
Value[Number]: GENERALNUMBER;
Value[String]: STRING;
Body[RegularExpression]: STRING;
Flags[RegularExpression]: STRING;
```

~~12.1~~11.2 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
```

Semantics

```
Name[Identifier]: STRING;
  Name[Identifier □ Identifier] = Name[Identifier];
  Name[Identifier □ get] = "get";
  Name[Identifier □ set] = "set";
```

11.3 Qualified Identifiers

Syntax

```
SimpleQualifiedIdentifier □
    Identifier
  | Identifier :: Identifier
  | ReservedNamespace :: Identifier

ExpressionQualifiedIdentifier □ ParenExpression :: Identifier

QualifiedIdentifier □
    SimpleQualifiedIdentifier
  | ExpressionQualifiedIdentifier
```

Validation

```
OpenNamespaces[SimpleQualifiedIdentifier]: NAMESPACE{};

Strict[SimpleQualifiedIdentifier]: BOOLEAN;

proc Validate[SimpleQualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
  [SimpleQualifiedIdentifier □ Identifier] do
    OpenNamespaces[SimpleQualifiedIdentifier] □ cxt.openNamespaces;
    Strict[SimpleQualifiedIdentifier] □ cxt.strict;
    [SimpleQualifiedIdentifier □ Identifier :: Identifier] do
        OpenNamespaces[SimpleQualifiedIdentifier] □ cxt.openNamespaces;
    [SimpleQualifiedIdentifier □ ReservedNamespace :: Identifier] do
        Evaluate Validate[ReservedNamespace](cxt, env) and ignore its result
end proc;

Strict[ExpressionQualifiedIdentifier]: BOOLEAN;

proc Validate[ExpressionQualifiedIdentifier □ ParenExpression :: Identifier] (cxt: CONTEXT, env: ENVIRONMENT)
    Strict[ExpressionQualifiedIdentifier] □ cxt.strict;
    Evaluate Validate[ParenExpression](cxt, env) and ignore its result
end proc;

Strict[QualifiedIdentifier]: BOOLEAN;

proc Validate[QualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
    [QualifiedIdentifier □ SimpleQualifiedIdentifier] do
        Strict[QualifiedIdentifier] □ cxt.strict;
        Evaluate Validate[SimpleQualifiedIdentifier](cxt, env) and ignore its result;
    [QualifiedIdentifier □ ExpressionQualifiedIdentifier] do
        Strict[QualifiedIdentifier] □ cxt.strict;
        Evaluate Validate[ExpressionQualifiedIdentifier](cxt, env) and ignore its result
end proc;
```

Setup

```
proc Setup[SimpleQualifiedIdentifier] ()
    [SimpleQualifiedIdentifier □ Identifier] do nothing;
    [SimpleQualifiedIdentifier □ Identifier :: Identifier] do nothing;
    [SimpleQualifiedIdentifier □ ReservedNamespace :: Identifier] do
        Evaluate Setup[ReservedNamespace]() and ignore its result
end proc;

proc Setup[ExpressionQualifiedIdentifier □ ParenExpression :: Identifier] ()
    Evaluate Setup[ParenExpression]() and ignore its result
end proc;
```

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

Evaluation

```
proc Eval[SimpleQualifiedIdentifier] (env: ENVIRONMENT, phase: PHASE): MULTINAME
    [SimpleQualifiedIdentifier □ Identifier] do
        return {ns::(Name[Identifier]) | □ns □ OpenNamespaces[SimpleQualifiedIdentifier]};
    [SimpleQualifiedIdentifier □ Identifier_1 :: Identifier_2] do
        multiname: MULTINAME □ {ns::(Name[Identifier_1]) | □ns □ OpenNamespaces[SimpleQualifiedIdentifier]};
        a: OBJECT □ lexicalRead(env, multiname, phase);
        if a □ NAMESPACE then
            throw a TypeError exception — the qualifier must be a namespace
        end if;
        return {a::(Name[Identifier_2])};
    [SimpleQualifiedIdentifier □ ReservedNamespace :: Identifier] do
        q: NAMESPACE □ Eval[ReservedNamespace](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 a TypeError exception — the qualifier must be a namespace
    end if;
    return {q::(Name[Identifier])}
end proc;
```

Eval[*QualifiedIdentifier*] (*env*: ENVIRONMENT, *phase*: PHASE): MULTINAME propagates the call to Eval to nonterminals in the expansion of *QualifiedIdentifier*.

11.4 Primary Expressions

Syntax

```
PrimaryExpression □
      null
   |  true
   |  false
   |  Number
   |  String
   |  this
   |  RegularExpression
   |  ReservedNamespace
   |  ParenListExpression
   |  ArrayLiteral
   |  ObjectLiteral
   |  FunctionExpression

ReservedNamespace □
      public
   |  private

ParenExpression □ ( AssignmentExpression^allowIn )

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

Validation

```
proc Validate[PrimaryExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [PrimaryExpression □ null] do nothing;
    [PrimaryExpression □ true] do nothing;
    [PrimaryExpression □ false] do nothing;
    [PrimaryExpression □ Number] do nothing;
    [PrimaryExpression □ String] do nothing;
    [PrimaryExpression □ this] do
        frame: PARAMETERFRAMEOPT □ getEnclosingParameterFrame(env);
        if frame = none then
            if cxt.strict then
                throw a SyntaxError exception — this can be used outside a function only in non-strict mode
            end if
        elsif frame.kind = plainFunction then
            throw a SyntaxError exception — this function does not define this
        end if;
    [PrimaryExpression □ RegularExpression] do nothing;
    [PrimaryExpression □ ReservedNamespace] do
        Evaluate Validate[ReservedNamespace](cxt, env) and ignore its result;
    [PrimaryExpression □ ParenListExpression] do
        Evaluate Validate[ParenListExpression](cxt, env) and ignore its result;
    [PrimaryExpression □ ArrayLiteral] do
        Evaluate Validate[ArrayLiteral](cxt, env) and ignore its result;
    [PrimaryExpression □ ObjectLiteral] do
        Evaluate Validate[ObjectLiteral](cxt, env) and ignore its result;
    [PrimaryExpression □ FunctionExpression] do
        Evaluate Validate[FunctionExpression](cxt, env) and ignore its result
end proc;
```

```
proc Validate[ReservedNamespace] (cxt: CONTEXT, env: ENVIRONMENT)
    [ReservedNamespace □ public] do nothing;
    [ReservedNamespace □ private] do
        if getEnclosingClass(env) = none then
            throw a SyntaxError exception — private is meaningful only inside a class
        end if
end proc;
```

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

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

Setup

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

```
proc Setup[ReservedNamespace] ()
    [ReservedNamespace □ public] do nothing;
    [ReservedNamespace □ private] do nothing
end proc;
```

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

Setup[*ParenListExpression*] () propagates the call to Setup to nonterminals 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 ⇒ Number] do return Value[Number];
    [PrimaryExpression ⇒ String] do return Value[String];
    [PrimaryExpression ⇒ this] do
        frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
        if frame = none then return getPackageFrame(env) end if;
        note Validate ensured that frame.kind ≠ plainFunction at this point.
        this: OBJECTOPT ← frame.this;
        if this = none then
            note If Validate passed, this can be uninitialised only when phase = compile.
            throw a ConstantError exception — a constant expression cannot read an uninitialised this parameter
        end if;
        if not frame.superconstructorCalled then
            throw an UninitializedError exception — can’t access this from within a constructor before the
                superconstructor has been called
        end if;
        return this;
    [PrimaryExpression ⇒ RegularExpression] do
        return Body[RegularExpression] ⊕ “#” ⊕ Flags[RegularExpression];
    [PrimaryExpression ⇒ ReservedNamespace] do
        return Eval[ReservedNamespace](env, phase);
    [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[ReservedNamespace] (env: ENVIRONMENT, phase: PHASE): NAMESPACE
    [ReservedNamespace ⇒ public] do return public;
    [ReservedNamespace ⇒ private] do
        c: CLASSOPT ← getEnclosingClass(env);
        note Validate already ensured that c ≠ none.
        return c.privateNamespace
end proc;
```

Eval[*ParenExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF propagates the call to Eval to nonterminals in the expansion of *ParenExpression*.

```
proc Eval[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ParenListExpression → ParenExpression] do return Eval[ParenExpression](env, phase);
  [ParenListExpression → ( ListExpression^allowIn , AssignmentExpression^allowIn )] do
    Evaluate readReference(Eval[ListExpression^allowIn](env, phase), phase) and ignore its result;
    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;
```

11.5 Function Expressions

Syntax

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

Validation

```
F[FunctionExpression]: UNINSTANTIATEDFUNCTION;

proc Validate[FunctionExpression] (cxt: CONTEXT, env: ENVIRONMENT)
  [FunctionExpression → function FunctionCommon] do
    kind: STATICFUNCTIONKIND ← plainFunction;
    if not cxt.strict and Plain[FunctionCommon] then kind ← uncheckedFunction
    end if;
    F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](cxt, env, kind);
  [FunctionExpression → function Identifier FunctionCommon] do
    v: VARIABLE ← new VARIABLE⟨⟨type: Function, value: none, immutable: true, setup: none, initializer: busy⟩⟩
    b: LOCALBINDING ← LOCALBINDING⟨⟨qname: public::(Name[Identifier]), accesses: readWrite, explicit: false,
        enumerable: true, content: v⟩⟩
    compileFrame: LOCALFRAME ← new LOCALFRAME⟨⟨localBindings: {b}⟩⟩
    kind: STATICFUNCTIONKIND ← plainFunction;
    if not cxt.strict and Plain[FunctionCommon] then kind ← uncheckedFunction
    end if;
    F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](cxt, [compileFrame] ⊕ env, kind)
end proc;
```

Setup

```
proc Setup[FunctionExpression] ()
  [FunctionExpression → function FunctionCommon] do
    Evaluate Setup[FunctionCommon]() and ignore its result;
    [FunctionExpression → function Identifier FunctionCommon] do
      Evaluate Setup[FunctionCommon]() and ignore its result
  end proc;
```

Evaluation

```
  proc Eval[FunctionExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [FunctionExpression → function FunctionCommon] do
      if phase = compile then
        throw a ConstantError exception — a function expression is not a constant expression because it can
              evaluate to different values
      end if;
      return instantiateFunction(F[FunctionExpression], env);
    [FunctionExpression → function Identifier FunctionCommon] do
      if phase = compile then
        throw a ConstantError exception — a function expression is not a constant expression because it can
              evaluate to different values
      end if;
      v: VARIABLE ← new VARIABLE⟨type: Function, value: none, immutable: true, setup: none, initializer: none⟩
      b: LOCALBINDING ← LOCALBINDING⟨qname: public::(Name[Identifier]), accesses: readWrite, explicit: false,
            enumerable: true, content: v⟩
      runtimeFrame: LOCALFRAME ← new LOCALFRAME⟨localBindings: {b}⟩
      f: SIMPLEINSTANCE ← instantiateFunction(F[FunctionExpression], [runtimeFrame] ⊕ env);
      v.value ← f;
      return f
  end proc;
```

11.6 Object Literals

Syntax

```
ObjectLiteral → { FieldList }

FieldList →
      «empty»
   |  NonemptyFieldList

NonemptyFieldList →
      LiteralField
   |  LiteralField , NonemptyFieldList

LiteralField → FieldName : AssignmentExpression^allowIn

FieldName →
      QualifiedIdentifier
   |  String
   |  Number
   |  ParenExpression
```

Validation

Validate[ObjectLiteral] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of ObjectLiteral.

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

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

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

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

Setup

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

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

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

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

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

Evaluation

```
proc Eval[ObjectLiteral □ { FieldList }] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  if phase = compile then
    throw a ConstantError exception — an object literal is not a constant expression because it evaluates to a new
        object each time it is evaluated
  end if;
  o: OBJECT □ construct(Object, [], phase);
  Evaluate Eval[FieldList](env, o, phase) and ignore its result;
  return o
end proc;
```

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

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

```
proc Eval[LiteralField □ FieldName : AssignmentExpression^allowIn] (env: ENVIRONMENT, o: OBJECT, phase: {run})
  multiname: MULTINAME □ Eval[FieldName](env, phase);
  value: OBJECT □ readReference(Eval[AssignmentExpression^allowIn](env, phase), phase);
  Evaluate dotWrite(o, multiname, value, phase) and ignore its result
end proc;

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

11.7 Array Literals

Syntax

```
ArrayLiteral  [ ElementList ]

ElementList 
    «empty»
  | LiteralElement
  | , ElementList
  | LiteralElement , ElementList

LiteralElement  AssignmentExpression^allowIn
```

Validation

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

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

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

Setup

Setup[*ArrayLiteral*] () propagates the call to Setup to nonterminals in the expansion of *ArrayLiteral*.

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

Setup[*LiteralElement*] () propagates the call to Setup to nonterminals in the expansion of *LiteralElement*.

Evaluation

```
proc Eval[ArrayLiteral  [ ElementList ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    if phase = compile then
        throw a ConstantError exception — an array literal is not a constant expression because it evaluates to a new object
            each time it is evaluated
    end if;
    o: OBJECT  construct(Array, [], phase);
    length: INTEGER  Eval[ElementList](env, 0, o, phase);
    Evaluate writeArrayPrivateLength(o, length, phase) and ignore its result;
    return o
end proc;

proc Eval[ElementList] (env: ENVIRONMENT, length: INTEGER, o: OBJECT, phase: {run}): INTEGER
    [ElementList  «empty»] do return length;
    [ElementList  LiteralElement] do
        Evaluate Eval[LiteralElement](env, length, o, phase) and ignore its result;
        return length + 1;
    [ElementList_0  , ElementList_1] do
        return Eval[ElementList_1](env, length + 1, o, phase);
    [ElementList0 ⇒ LiteralElement , ElementList1] do
      Evaluate Eval[LiteralElement](env, length, o, phase) and ignore its result;
      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);
   Evaluate indexWrite(o, length, value, phase) and ignore its result
end proc;
```

11.8 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 then
         throw a SyntaxError exception — a super expression is meaningful only inside a class
      end if;
      frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
      if frame = none or frame.kind ≠ STATICFUNCTIONKIND then
         throw a SyntaxError exception — a super expression without an argument is meaningful only inside an
               instance method or a constructor
      end if;
      if c.super = none then
         throw a SyntaxError exception — a super expression is meaningful only if the enclosing class has a superclass
      end if;
   [SuperExpression ⇒ super ParenExpression] do
      c: CLASSOPT ← getEnclosingClass(env);
      if c = none then
         throw a SyntaxError exception — a super expression is meaningful only inside a class
      end if;
      if c.super = none then
         throw a SyntaxError exception — a super expression is meaningful only if the enclosing class has a superclass
      end if;
      Evaluate Validate[ParenExpression](cxt, env) and ignore its result
end proc;
```

Setup

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

Evaluation

```
proc Eval[SuperExpression] (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT
  [SuperExpression □ super] do
    frame: PARAMETERFRAMEOPT □ getEnclosingParameterFrame(env);
    note Validate ensured that frame ≠ none and frame.kind □ STATICFUNCTIONKIND at this point.
    this: OBJECTOPT □ frame.this;
    if this = none then
      note If Validate passed, this can be uninitialised only when phase = compile.
      throw a ConstantError exception — a constant expression cannot read an uninitialised this parameter
    end if;
    if not frame.superconstructorCalled then
      throw an UninitializedError exception — can't access super from within a constructor before the
          superconstructor has been called
    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 □ coerce(o, limit);
  if coerced = null then return null end if;
  return LIMITEDINSTANCE□instance: coerced, limit: limit□
end proc;
```

11.9 Postfix Expressions

Syntax

```
PostfixExpression □
    AttributeExpression
  | FullPostfixExpression
  | ShortNewExpression

AttributeExpression □
    SimpleQualifiedIdentifier
  | AttributeExpression PropertyOperator
  | AttributeExpression Arguments

FullPostfixExpression □
    PrimaryExpression
  | ExpressionQualifiedIdentifier
  | FullNewExpression
  | FullPostfixExpression PropertyOperator
  | SuperExpression PropertyOperator
  | FullPostfixExpression Arguments
  | PostfixExpression [no line break] ++
  | PostfixExpression [no line break] --

FullNewExpression □ new FullNewSubexpression Arguments
```

FullNewSubexpression □
  PrimaryExpression
| *QualifiedIdentifier*
| *FullNewExpression*
| *FullNewSubexpression PropertyOperator*
| *SuperExpression PropertyOperator*

ShortNewExpression □ **new** *ShortNewSubexpression*

ShortNewSubexpression □
  FullNewSubexpression
| *ShortNewExpression*

Validation

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

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

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

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

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

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

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

Setup

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

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

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

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

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

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

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

Evaluation

Eval[*PostfixExpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF propagates the call to Eval to nonterminals in the expansion of *PostfixExpression*.

```
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[SimpleQualifiedIdentifier]□;
    [AttributeExpression_0 □ AttributeExpression_1 PropertyOperator] do
        a: OBJECT □ readReference(Eval[AttributeExpression_1](env, phase), phase);
        return Eval[PropertyOperator](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[ExpressionQualifiedIdentifier]□;
    [FullPostfixExpression □ FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    [FullPostfixExpression_0 □ FullPostfixExpression_1 PropertyOperator] do
        a: OBJECT □ readReference(Eval[FullPostfixExpression_1](env, phase), phase);
        return Eval[PropertyOperator](env, a, phase);
    [FullPostfixExpression □ SuperExpression PropertyOperator] do
        a: OBJOPTIONALLIMIT □ Eval[SuperExpression](env, phase);
        return Eval[PropertyOperator](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 a ConstantError exception — ++ cannot be used in a constant expression
        end if;
        r: OBJORREF  Eval[PostfixExpression](env, phase);
        a: OBJECT  readReference(r, phase);
        b: OBJECT  plus(a, phase);
        c: OBJECT  add(b, 1f64, phase);
        Evaluate writeReference(r, c, phase) and ignore its result;
        return b;
    [FullPostfixExpression  PostfixExpression [no line break] --] do
        if phase = compile then
            throw a ConstantError exception — -- cannot be used in a constant expression
        end if;
        r: OBJORREF  Eval[PostfixExpression](env, phase);
        a: OBJECT  readReference(r, phase);
        b: OBJECT  plus(a, phase);
        c: OBJECT  subtract(b, 1f64, phase);
        Evaluate writeReference(r, c, phase) and ignore its result;
        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 LEXICALREFERENCEenv: env, variableMultiname: m, strict: Strict[QualifiedIdentifier]
    [FullNewSubexpression  FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    [FullNewSubexpression0  FullNewSubexpression1 PropertyOperator] do
        a: OBJECT  readReference(Eval[FullNewSubexpression1](env, phase), phase);
        return Eval[PropertyOperator](env, a, phase);
    [FullNewSubexpression  SuperExpression PropertyOperator] do
        a: OBJOPTIONALLIMIT  Eval[SuperExpression](env, phase);
        return Eval[PropertyOperator](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;
```

Eval[*ShortNewSubexpression*] (*env*: ENVIRONMENT, *phase*: PHASE): OBJORREF propagates the call to Eval to nonterminals in the expansion of *ShortNewSubexpression*.

11.10 Property Operators

Syntax

PropertyOperator □
  . *QualifiedIdentifier*
| *Brackets*

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

Arguments □
  ( )
| *ParenListExpression*
| (*ExpressionsWithRest*)

ExpressionsWithRest □
  RestExpression
| *ListExpression*$^{\text{allowIn}}$, *RestExpression*

RestExpression □ ... *AssignmentExpression*$^{\text{allowIn}}$

Validation

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

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

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

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

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

Setup

Setup[*PropertyOperator*] () propagates the call to Setup to nonterminals in the expansion of *PropertyOperator*.

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

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

Setup[*ExpressionsWithRest*] () propagates the call to Setup to nonterminals in the expansion of *ExpressionsWithRest*.

Setup[*RestExpression*] () propagates the call to Setup to nonterminals in the expansion of *RestExpression*.

Evaluation

```
proc Eval[PropertyOperator] (env: ENVIRONMENT, base: OBJOPTIONALLIMIT, phase: PHASE): OBJORREF
    [PropertyOperator □ . QualifiedIdentifier] do
        m: MULTINAME □ Eval[QualifiedIdentifier](env, phase);
        case base of
            OBJECT do
                return DOTREFERENCE□base: base, limit: objectType(base), multiname: m□
            LIMITEDINSTANCE do
                return DOTREFERENCE□base: base.instance, limit: base.limit, multiname: m□
        end case;
    [PropertyOperator □ Brackets] do
        args: OBJECT[] □ Eval[Brackets](env, phase);
        case base of
            OBJECT do
                return BRACKETREFERENCE□base: base, limit: objectType(base), args: args□
            LIMITEDINSTANCE do
                return BRACKETREFERENCE□base: 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);
    [Brackets □ [ ExpressionsWithRest ]] do return Eval[ExpressionsWithRest](env, phase)
end proc;

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

proc Eval[ExpressionsWithRest] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
    [ExpressionsWithRest □ RestExpression] do return Eval[RestExpression](env, phase);
    [ExpressionsWithRest □ ListExpression^allowIn , RestExpression] do
        args1: OBJECT[] □ EvalAsList[ListExpression^allowIn](env, phase);
        args2: OBJECT[] □ Eval[RestExpression](env, phase);
        return args1 ⊕ args2
end proc;
```

```
proc Eval[RestExpression → ... AssignmentExpression^allowIn] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
    a: OBJECT ← readReference(Eval[AssignmentExpression^allowIn](env, phase), phase);
    length: INTEGER ← readLength(a, phase);
    i: INTEGER ← 0;
    args: OBJECT[] ← [];
    while i ≠ length do
        arg: OBJECTOPT ← indexRead(a, i, phase);
        if arg = none then
            An implementation may, at its discretion, either throw a ReferenceError or treat the hole as a missing argument,
            substituting the called function's default parameter value if there is one, undefined if the called function is
            unchecked, or throwing an ArgumentError exception otherwise. An implementation must not replace such a hole
            with undefined except when the called function is unchecked or happens to have undefined as its default
            parameter value.
        end if;
        args ← args ⊕ [arg];
        i ← i + 1
    end while;
    return args
end proc;
```

11.11 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
        Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
    [UnaryExpression → delete PostfixExpression] do
        Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
        Strict[UnaryExpression] ← cxt.strict;
    [UnaryExpression₀ → void UnaryExpression₁] do
        Evaluate Validate[UnaryExpression₁](cxt, env) and ignore its result;
    [UnaryExpression₀ → typeof UnaryExpression₁] do
        Evaluate Validate[UnaryExpression₁](cxt, env) and ignore its result;
    [UnaryExpression → ++ PostfixExpression] do
        Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
    [UnaryExpression □ -- PostfixExpression] do
      Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
    [$UnaryExpression_0$ □ + $UnaryExpression_1$] do
      Evaluate Validate[$UnaryExpression_1$](cxt, env) and ignore its result;
    [$UnaryExpression_0$ □ - $UnaryExpression_1$] do
      Evaluate Validate[$UnaryExpression_1$](cxt, env) and ignore its result;
    [UnaryExpression □ - NegatedMinLong] do nothing;
    [$UnaryExpression_0$ □ ~ $UnaryExpression_1$] do
      Evaluate Validate[$UnaryExpression_1$](cxt, env) and ignore its result;
    [$UnaryExpression_0$ □ ! $UnaryExpression_1$] do
      Evaluate Validate[$UnaryExpression_1$](cxt, env) and ignore its result
  end proc;
```

Setup

Setup[*UnaryExpression*] () propagates the call to Setup to nonterminals 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 a ConstantError exception — delete cannot be used in a constant expression
      end if;
      r: OBJORREF □ Eval[PostfixExpression](env, phase);
      return deleteReference(r, Strict[UnaryExpression], phase);
    [$UnaryExpression_0$ □ void $UnaryExpression_1$] do
      Evaluate readReference(Eval[$UnaryExpression_1$](env, phase), phase) and ignore its result;
      return undefined;
    [$UnaryExpression_0$ □ typeof $UnaryExpression_1$] do
      a: OBJECT □ readReference(Eval[$UnaryExpression_1$](env, phase), phase);
      c: CLASS □ objectType(a);
      return c.typeofString;
    [UnaryExpression □ ++ PostfixExpression] do
      if phase = compile then
        throw a ConstantError exception — ++ cannot be used in a constant expression
      end if;
      r: OBJORREF □ Eval[PostfixExpression](env, phase);
      a: OBJECT □ readReference(r, phase);
      b: OBJECT □ plus(a, phase);
      c: OBJECT □ add(b, $1_{f64}$, phase);
      Evaluate writeReference(r, c, phase) and ignore its result;
      return c;
    [UnaryExpression → -- PostfixExpression] do
       if phase = compile then
          throw a ConstantError exception — -- cannot be used in a constant expression
       end if;
       r: OBJORREF ← Eval[PostfixExpression](env, phase);
       a: OBJECT ← readReference(r, phase);
       b: OBJECT ← plus(a, phase);
       c: OBJECT ← subtract(b, $1_{f64}$, phase);
       Evaluate writeReference(r, c, phase) and ignore its result;
       return c;
    [$UnaryExpression_0$ → + $UnaryExpression_1$] do
       a: OBJECT ← readReference(Eval[$UnaryExpression_1$](env, phase), phase);
       return plus(a, phase);
    [$UnaryExpression_0$ → - $UnaryExpression_1$] do
       a: OBJECT ← readReference(Eval[$UnaryExpression_1$](env, phase), phase);
       return minus(a, phase);
    [UnaryExpression → - NegatedMinLong] do return $(-2^{63})_{long}$;
    [$UnaryExpression_0$ → ~ $UnaryExpression_1$] do
       a: OBJECT ← readReference(Eval[$UnaryExpression_1$](env, phase), phase);
       return bitNot(a, phase);
    [$UnaryExpression_0$ → ! $UnaryExpression_1$] do
       a: OBJECT ← readReference(Eval[$UnaryExpression_1$](env, phase), phase);
       return logicalNot(a, phase)
 end proc;
```

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

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

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

```
 proc minus(a: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER ← objectToGeneralNumber(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 □ objectToGeneralNumber(a, phase);
    case x of
        LONG do i: {$-2^{63}$ ... $2^{63} - 1$} □ x.value; return (bitwiseXor(i, –1))$_{\text{long}}$;
        ULONG do
            i: {0 ... $2^{64} - 1$} □ x.value;
            return (bitwiseXor(i, 0xFFFFFFFFFFFFFFFF))$_{\text{ulong}}$;
        FLOAT32 □ FLOAT64 do
            i: {$-2^{31}$ ... $2^{31} - 1$} □ signedWrap32(truncateToInteger(x));
            return (bitwiseXor(i, –1))$_{\text{f64}}$
    end case
end proc;
```

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

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

11.12 Multiplicative Operators

Syntax

```
MultiplicativeExpression □
        UnaryExpression
    |   MultiplicativeExpression * UnaryExpression
    |   MultiplicativeExpression / UnaryExpression
    |   MultiplicativeExpression % UnaryExpression
```

Validation

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

Setup

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

Evaluation

```
proc Eval[MultiplicativeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [MultiplicativeExpression □ UnaryExpression] do
        return Eval[UnaryExpression](env, phase);
    [MultiplicativeExpression$_0$ □ MultiplicativeExpression$_1$ * UnaryExpression] do
        a: OBJECT □ readReference(Eval[MultiplicativeExpression$_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[UnaryExpression](env, phase), phase);
        return multiply(a, b, phase);
    [MultiplicativeExpression$_0$ □ MultiplicativeExpression$_1$ / UnaryExpression] do
        a: OBJECT □ readReference(Eval[MultiplicativeExpression$_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[UnaryExpression](env, phase), phase);
        return divide(a, b, phase);
    [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 ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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 ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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 ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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 ≥ 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;
```

11.13 Additive Operators

Syntax

```
AdditiveExpression →
     MultiplicativeExpression
  |  AdditiveExpression + MultiplicativeExpression
  |  AdditiveExpression - MultiplicativeExpression
```

Validation

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

Setup

Setup[*AdditiveExpression*] () propagates the call to Setup to nonterminals 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 ← objectToPrimitive(a, none, phase);
  bp: PRIMITIVEOBJECT ← objectToPrimitive(b, none, phase);
  if ap ∈ CHAR16 ∪ STRING or bp ∈ CHAR16 ∪ STRING then
    return objectToString(ap, phase) ⊕ objectToString(bp, phase)
  end if;
  x: GENERALNUMBER ← objectToGeneralNumber(ap, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(bp, phase);
  if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
    i: INTEGEROPT ← checkInteger(x);
    j: INTEGEROPT ← checkInteger(y);
    if $i \neq$ none and $j \neq$ none then
      k: INTEGER ← $i + j$;
      if x ∈ ULONG or y ∈ ULONG then return integerToULong(k)
      else return integerToLong(k)
      end if
    end if
  end if;
  return float64Add(toFloat64(x), toFloat64(y))
end proc;
```

```
proc subtract(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER □ objectToGeneralNumber(a, phase);
    y: GENERALNUMBER □ objectToGeneralNumber(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;
```

11.14 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 nonterminals in the expansion of *ShiftExpression*.

Setup

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

Evaluation

```
proc Eval[ShiftExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ShiftExpression □ AdditiveExpression] do
        return Eval[AdditiveExpression](env, phase);
    [ShiftExpression$_0$ □ ShiftExpression$_1$ << AdditiveExpression] do
        a: OBJECT □ readReference(Eval[ShiftExpression$_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[AdditiveExpression](env, phase), phase);
        return shiftLeft(a, b, phase);
    [ShiftExpression$_0$ □ ShiftExpression$_1$ >> AdditiveExpression] do
        a: OBJECT □ readReference(Eval[ShiftExpression$_1$](env, phase), phase);
        b: OBJECT □ readReference(Eval[AdditiveExpression](env, phase), phase);
        return shiftRight(a, b, phase);
    [ShiftExpression$_0$ □ ShiftExpression$_1$ >>> AdditiveExpression] do
        a: OBJECT □ readReference(Eval[ShiftExpression$_1$](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 □ objectToGeneralNumber(a, phase);
    count: INTEGER □ truncateToInteger(objectToGeneralNumber(b, phase));
    case x of
        FLOAT32 □ FLOAT64 do
            count □ bitwiseAnd(count, 0x1F);
            i: $\{-2^{31} \ldots 2^{31} - 1\}$ □ signedWrap32(bitwiseShift(truncateToInteger(x), count));
            return $i_{f64}$;
        LONG do
            count □ bitwiseAnd(count, 0x3F);
            i: $\{-2^{63} \ldots 2^{63} - 1\}$ □ signedWrap64(bitwiseShift(x.value, count));
            return $i_{long}$;
        ULONG do
            count □ bitwiseAnd(count, 0x3F);
            i: $\{0 \ldots 2^{64} - 1\}$ □ unsignedWrap64(bitwiseShift(x.value, count));
            return $i_{ulong}$
    end case
end proc;

proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
    x: GENERALNUMBER □ objectToGeneralNumber(a, phase);
    count: INTEGER □ truncateToInteger(objectToGeneralNumber(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 $i_{f64}$;
        LONG do
            count □ bitwiseAnd(count, 0x3F);
            i: $\{-2^{63} \ldots 2^{63} - 1\}$ □ bitwiseShift(x.value, –count);
            return $i_{long}$;
        ULONG do
            count □ bitwiseAnd(count, 0x3F);
            i: $\{-2^{63} \ldots 2^{63} - 1\}$ □ bitwiseShift(signedWrap64(x.value), –count);
            return $(unsignedWrap64(i))_{ulong}$
    end case
end proc;
```

```
proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
   x: GENERALNUMBER □ objectToGeneralNumber(a, phase);
   count: INTEGER □ truncateToInteger(objectToGeneralNumber(b, phase));
   case x of
      FLOAT32 □ FLOAT64 do
         i: {0 ... $2^{32}$ – 1} □ unsignedWrap32(truncateToInteger(x));
         count □ bitwiseAnd(count, 0x1F);
         i □ bitwiseShift(i, –count);
         return $i_{f64}$;
      LONG do
         count □ bitwiseAnd(count, 0x3F);
         i: {0 ... $2^{64}$ – 1} □ bitwiseShift(unsignedWrap64(x.value), –count);
         return $(signedWrap64(i))_{long}$;
      ULONG do
         count □ bitwiseAnd(count, 0x3F);
         i: {0 ... $2^{64}$ – 1} □ bitwiseShift(x.value, –count);
         return $i_{ulong}$
   end case
end proc;
```

11.15 Relational Operators

Syntax

RelationalExpression^{allowIn} □
   ShiftExpression
 | *RelationalExpression*^{allowIn} **<** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **>** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **<=** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **>=** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **is** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **as** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **in** *ShiftExpression*
 | *RelationalExpression*^{allowIn} **instanceof** *ShiftExpression*

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

Validation

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

Setup

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

Evaluation

```
proc Eval[RelationalExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [RelationalExpression^β ⇒ ShiftExpression] do
      return Eval[ShiftExpression](env, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 < ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      return isLess(a, b, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 > ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      return isLess(b, a, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 <= ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      return isLessOrEqual(a, b, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 >= ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      return isLessOrEqual(b, a, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 is ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      c: CLASS ← objectToClass(b);
      return is(a, c);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 as ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      c: CLASS ← objectToClass(b);
      return coerceOrNull(a, c);
   [RelationalExpression^allowIn_0 ⇒ RelationalExpression^allowIn_1 in ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^allowIn_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      return hasProperty(b, a, false, phase);
   [RelationalExpression^β_0 ⇒ RelationalExpression^β_1 instanceof ShiftExpression] do
      a: OBJECT ← readReference(Eval[RelationalExpression^β_1](env, phase), phase);
      b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
      if b ∈ CLASS then return is(a, b)
      elsif is(b, PrototypeFunction) then
         prototype: OBJECT ← dotRead(b, {public::"prototype"}, phase);
         return prototype ∈ archetypes(a)
      else throw a TypeError exception
      end if
end proc;
```

```
proc isLess(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    ap: PRIMITIVEOBJECT □ objectToPrimitive(a, hintNumber, phase);
    bp: PRIMITIVEOBJECT □ objectToPrimitive(b, hintNumber, phase);
    if ap □ CHAR16 □ STRING and bp □ CHAR16 □ STRING then
        return toString(ap) < toString(bp)
    end if;
    return generalNumberCompare(objectToGeneralNumber(ap, phase), objectToGeneralNumber(bp, phase)) = less
end proc;

proc isLessOrEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
    ap: PRIMITIVEOBJECT □ objectToPrimitive(a, hintNumber, phase);
    bp: PRIMITIVEOBJECT □ objectToPrimitive(b, hintNumber, phase);
    if ap □ CHAR16 □ STRING and bp □ CHAR16 □ STRING then
        return toString(ap) ≤ toString(bp)
    end if;
    return generalNumberCompare(objectToGeneralNumber(ap, phase),
            objectToGeneralNumber(bp, phase)) □ {less, equal}
end proc;
```

11.16 Equality Operators

Syntax

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

Validation

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

Setup

Setup[*EqualityExpression*□] () propagates the call to Setup to nonterminals in the expansion of *EqualityExpression*□.

Evaluation

```
proc Eval[EqualityExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [EqualityExpression□ □ RelationalExpression□] do
        return Eval[RelationalExpression□](env, phase);
    [EqualityExpression□_0 □ EqualityExpression□_1 == RelationalExpression□] do
        a: OBJECT □ readReference(Eval[EqualityExpression□_1](env, phase), phase);
        b: OBJECT □ readReference(Eval[RelationalExpression□](env, phase), phase);
        return isEqual(a, b, phase);
    [EqualityExpression□_0 □ EqualityExpression□_1 != RelationalExpression□] do
        a: OBJECT □ readReference(Eval[EqualityExpression□_1](env, phase), phase);
        b: OBJECT □ readReference(Eval[RelationalExpression□](env, phase), phase);
        return not isEqual(a, b, phase);
    [EqualityExpression□0 □ EqualityExpression□1 === RelationalExpression□] do
        a: OBJECT □ readReference(Eval[EqualityExpression□1](env, phase), phase);
        b: OBJECT □ readReference(Eval[RelationalExpression□](env, phase), phase);
        return isStrictlyEqual(a, b, phase);
    [EqualityExpression□0 □ EqualityExpression□1 !== RelationalExpression□] do
        a: OBJECT □ readReference(Eval[EqualityExpression□1](env, phase), phase);
        b: OBJECT □ readReference(Eval[RelationalExpression□](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(objectToGeneralNumber(a, phase), b, phase)
            end if;
        GENERALNUMBER do
            bp: PRIMITIVEOBJECT □ objectToPrimitive(b, none, phase);
            case bp of
                UNDEFINED □ NULL do return false;
                BOOLEAN □ GENERALNUMBER □ CHAR16 □ STRING do
                    return generalNumberCompare(a, objectToGeneralNumber(bp, phase)) = equal
            end case;
        CHAR16 □ STRING do
            bp: PRIMITIVEOBJECT □ objectToPrimitive(b, none, phase);
            case bp of
                UNDEFINED □ NULL do return false;
                BOOLEAN □ GENERALNUMBER do
                    return generalNumberCompare(objectToGeneralNumber(a, phase),
                            objectToGeneralNumber(bp, phase)) = equal;
                CHAR16 □ STRING do return toString(a) = toString(bp)
            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 □ CHAR16 □ STRING do
                    ap: PRIMITIVEOBJECT □ objectToPrimitive(a, none, 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;
```

11.17 Binary Bitwise Operators

Syntax

```
BitwiseAndExpression^β ⇒
    EqualityExpression^β
  | BitwiseAndExpression^β & EqualityExpression^β

BitwiseXorExpression^β ⇒
    BitwiseAndExpression^β
  | BitwiseXorExpression^β ^ BitwiseAndExpression^β

BitwiseOrExpression^β ⇒
    BitwiseXorExpression^β
  | BitwiseOrExpression^β | BitwiseXorExpression^β
```

Validation

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

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

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

Setup

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

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

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

Evaluation

```
proc Eval[BitwiseAndExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [BitwiseAndExpression^β ⇒ EqualityExpression^β] do
    return Eval[EqualityExpression^β](env, phase);
  [BitwiseAndExpression^β_0 ⇒ BitwiseAndExpression^β_1 & EqualityExpression^β] do
    a: OBJECT ← readReference(Eval[BitwiseAndExpression^β_1](env, phase), phase);
    b: OBJECT ← readReference(Eval[EqualityExpression^β](env, phase), phase);
    return bitAnd(a, b, phase)
end proc;

proc Eval[BitwiseXorExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [BitwiseXorExpression^β ⇒ BitwiseAndExpression^β] do
    return Eval[BitwiseAndExpression^β](env, phase);
  [BitwiseXorExpression^β_0 ⇒ BitwiseXorExpression^β_1 ^ BitwiseAndExpression^β] do
    a: OBJECT ← readReference(Eval[BitwiseXorExpression^β_1](env, phase), phase);
    b: OBJECT ← readReference(Eval[BitwiseAndExpression^β](env, phase), phase);
    return bitXor(a, b, phase)
end proc;
```

```
proc Eval[BitwiseOrExpression^β] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [BitwiseOrExpression^β ⇒ BitwiseXorExpression^β] do
        return Eval[BitwiseXorExpression^β](env, phase);
    [BitwiseOrExpression^β_0 ⇒ BitwiseOrExpression^β_1 | BitwiseXorExpression^β] do
        a: OBJECT ← readReference(Eval[BitwiseOrExpression^β_1](env, phase), phase);
        b: OBJECT ← readReference(Eval[BitwiseXorExpression^β](env, phase), phase);
        return bitOr(a, b, phase)
end proc;

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
    x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
    y: GENERALNUMBER ← objectToGeneralNumber(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 $(bitwiseAnd(i, j))_{f64}$
    end if
end proc;

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
    x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
    y: GENERALNUMBER ← objectToGeneralNumber(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 $(bitwiseXor(i, j))_{f64}$
    end if
end proc;
```

```
proc bitOr(a: Object, b: Object, phase: Phase): GeneralNumber
    x: GeneralNumber ← objectToGeneralNumber(a, phase);
    y: GeneralNumber ← objectToGeneralNumber(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: {$-2^{63}$ ... $2^{63} - 1$} ← bitwiseOr(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 $(bitwiseOr(i, j))_{f64}$
    end if
end proc;
```

11.18 Binary Logical Operators

Syntax

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

LogicalXorExpression^β ⇒
    LogicalAndExpression^β
  | *LogicalXorExpression*^{β} **^^** *LogicalAndExpression*^{β}

LogicalOrExpression^β ⇒
    LogicalXorExpression^β
  | *LogicalOrExpression*^{β} **||** *LogicalXorExpression*^{β}

Validation

Validate[*LogicalAndExpression*^{β}] (*cxt*: Context, *env*: Environment) propagates the call to Validate to nonterminals in the expansion of *LogicalAndExpression*^{β}.

Validate[*LogicalXorExpression*^{β}] (*cxt*: Context, *env*: Environment) propagates the call to Validate to nonterminals in the expansion of *LogicalXorExpression*^{β}.

Validate[*LogicalOrExpression*^{β}] (*cxt*: Context, *env*: Environment) propagates the call to Validate to nonterminals in the expansion of *LogicalOrExpression*^{β}.

Setup

Setup[*LogicalAndExpression*^{β}] () propagates the call to Setup to nonterminals in the expansion of *LogicalAndExpression*^{β}.

Setup[*LogicalXorExpression*^{β}] () propagates the call to Setup to nonterminals in the expansion of *LogicalXorExpression*^{β}.

Setup[*LogicalOrExpression*^{β}] () propagates the call to Setup to nonterminals in the expansion of *LogicalOrExpression*^{β}.

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 objectToBoolean(a) 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 □ objectToBoolean(a);
    bb: BOOLEAN □ objectToBoolean(b);
    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 objectToBoolean(a) then return a
    else return readReference(Eval[LogicalXorExpression□](env, phase), phase)
    end if
end proc;
```

11.19 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 nonterminals in the expansion of *ConditionalExpression*□.

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

Setup

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

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

Evaluation

```
proc Eval[ConditionalExpression□] (env: Environment, phase: Phase): ObjOrRef
    [ConditionalExpression□ □ LogicalOrExpression□] do
        return Eval[LogicalOrExpression□](env, phase);
    [ConditionalExpression□ □ LogicalOrExpression□ ? AssignmentExpression□$_1$ : AssignmentExpression□$_2$] do
        a: Object □ readReference(Eval[LogicalOrExpression□](env, phase), phase);
        if objectToBoolean(a) then
            return readReference(Eval[AssignmentExpression□$_1$](env, phase), phase)
        else return readReference(Eval[AssignmentExpression□$_2$](env, phase), phase)
        end if
end proc;

proc Eval[NonAssignmentExpression□] (env: Environment, phase: Phase): ObjOrRef
    [NonAssignmentExpression□ □ LogicalOrExpression□] do
        return Eval[LogicalOrExpression□](env, phase);
    [NonAssignmentExpression□$_0$ □ LogicalOrExpression□ ? NonAssignmentExpression□$_1$ : NonAssignmentExpression□
        $_2$] do
        a: Object □ readReference(Eval[LogicalOrExpression□](env, phase), phase);
        if objectToBoolean(a) then
            return readReference(Eval[NonAssignmentExpression□$_1$](env, phase), phase)
        else return readReference(Eval[NonAssignmentExpression□$_2$](env, phase), phase)
        end if
end proc;
```

11.20 Assignment Operators

Syntax

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

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

Semantics

```
tag andEq;

tag xorEq;

tag orEq;
```

Validation

```
proc Validate[AssignmentExpression$^□$] (cxt: Context, env: Environment)
   [AssignmentExpression$^□$ □ ConditionalExpression$^□$] do
      Evaluate Validate[ConditionalExpression$^□$](cxt, env) and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression = AssignmentExpression$^□_1$] do
      Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
      Evaluate Validate[AssignmentExpression$^□_1$](cxt, env) and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression CompoundAssignment AssignmentExpression$^□_1$] do
      Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
      Evaluate Validate[AssignmentExpression$^□_1$](cxt, env) and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression LogicalAssignment AssignmentExpression$^□_1$] do
      Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
      Evaluate Validate[AssignmentExpression$^□_1$](cxt, env) and ignore its result
end proc;
```

Setup

```
proc Setup[AssignmentExpression$^□$] ()
   [AssignmentExpression$^□$ □ ConditionalExpression$^□$] do
      Evaluate Setup[ConditionalExpression$^□$]() and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression = AssignmentExpression$^□_1$] do
      Evaluate Setup[PostfixExpression]() and ignore its result;
      Evaluate Setup[AssignmentExpression$^□_1$]() and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression CompoundAssignment AssignmentExpression$^□_1$] do
      Evaluate Setup[PostfixExpression]() and ignore its result;
      Evaluate Setup[AssignmentExpression$^□_1$]() and ignore its result;
   [AssignmentExpression$^□_0$ □ PostfixExpression LogicalAssignment AssignmentExpression$^□_1$] do
      Evaluate Setup[PostfixExpression]() and ignore its result;
      Evaluate Setup[AssignmentExpression$^□_1$]() and ignore its result
end proc;
```

Evaluation

```
proc Eval[AssignmentExpression$^□$] (env: Environment, phase: Phase): ObjOrRef
   [AssignmentExpression$^□$ □ ConditionalExpression$^□$] do
      return Eval[ConditionalExpression$^□$](env, phase);
```

```
  [AssignmentExpression$^{□}_{0}$ □ PostfixExpression = AssignmentExpression$^{□}_{1}$] do
    if phase = compile then
      throw a ConstantError exception — assignment cannot be used in a constant expression
    end if;
    ra: ObjOrRef □ Eval[PostfixExpression](env, phase);
    b: Object □ readReference(Eval[AssignmentExpression$^{□}_{1}$](env, phase), phase);
    Evaluate writeReference(ra, b, phase) and ignore its result;
    return b;
  [AssignmentExpression$^{□}_{0}$ □ PostfixExpression CompoundAssignment AssignmentExpression$^{□}_{1}$] do
    if phase = compile then
      throw a ConstantError exception — assignment cannot be used in a constant expression
    end if;
    rLeft: ObjOrRef □ Eval[PostfixExpression](env, phase);
    oLeft: Object □ readReference(rLeft, phase);
    oRight: Object □ readReference(Eval[AssignmentExpression$^{□}_{1}$](env, phase), phase);
    result: Object □ Op[CompoundAssignment](oLeft, oRight, phase);
    Evaluate writeReference(rLeft, result, phase) and ignore its result;
    return result;
  [AssignmentExpression$^{□}_{0}$ □ PostfixExpression LogicalAssignment AssignmentExpression$^{□}_{1}$] do
    if phase = compile then
      throw a ConstantError exception — assignment cannot be used in a constant expression
    end if;
    rLeft: ObjOrRef □ Eval[PostfixExpression](env, phase);
    oLeft: Object □ readReference(rLeft, phase);
    bLeft: Boolean □ objectToBoolean(oLeft);
    result: Object □ oLeft;
    case Operator[LogicalAssignment] of
      {andEq} do
        if bLeft then
          result □ readReference(Eval[AssignmentExpression$^{□}_{1}$](env, phase), phase)
        end if;
      {xorEq} do
        bRight: Boolean □ objectToBoolean(readReference(Eval[AssignmentExpression$^{□}_{1}$](env, phase), phase));
        result □ bLeft xor bRight;
      {orEq} do
        if not bLeft then
          result □ readReference(Eval[AssignmentExpression$^{□}_{1}$](env, phase), phase)
        end if
    end case;
    Evaluate writeReference(rLeft, result, phase) and ignore its result;
    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;
```

11.21 Comma Expressions

Syntax

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

Validation

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

Setup

Setup[*ListExpression*□] () propagates the call to Setup to nonterminals 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
        Evaluate readReference(Eval[ListExpression□_1](env, phase), phase) and ignore its result;
        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;
```

11.22 Type Expressions

Syntax

TypeExpression *NonAssignmentExpression*

Validation

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

Setup and Evaluation

```
proc SetupAndEval[TypeExpression   NonAssignmentExpression] (env: ENVIRONMENT): CLASS
    Evaluate Setup[NonAssignmentExpression]() and ignore its result;
    o: OBJECT   readReference(Eval[NonAssignmentExpression](env, compile), compile);
    return objectToClass(o)
end proc;
```

12 Statements

Syntax

  {abbrev, noShortIf, full}

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

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

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

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

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

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

Semicolon$^{full}$ □
      ;
   |  VirtualSemicolon
```

Validation

```
proc Validate[Statement$^{□}$] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS, preinst: BOOLEAN)
   [Statement$^{□}$ □ ExpressionStatement Semicolon$^{□}$] do
      Evaluate Validate[ExpressionStatement](cxt, env) and ignore its result;
   [Statement$^{□}$ □ SuperStatement Semicolon$^{□}$] do
      Evaluate Validate[SuperStatement](cxt, env) and ignore its result;
   [Statement$^{□}$ □ Block] do
      Evaluate Validate[Block](cxt, env, jt, preinst) and ignore its result;
   [Statement$^{□}$ □ LabeledStatement$^{□}$] do
      Evaluate Validate[LabeledStatement$^{□}$](cxt, env, sl, jt) and ignore its result;
   [Statement$^{□}$ □ IfStatement$^{□}$] do
      Evaluate Validate[IfStatement$^{□}$](cxt, env, jt) and ignore its result;
   [Statement$^{□}$ □ SwitchStatement] do
      Evaluate Validate[SwitchStatement](cxt, env, jt) and ignore its result;
   [Statement$^{□}$ □ DoStatement Semicolon$^{□}$] do
      Evaluate Validate[DoStatement](cxt, env, sl, jt) and ignore its result;
   [Statement$^{□}$ □ WhileStatement$^{□}$] do
      Evaluate Validate[WhileStatement$^{□}$](cxt, env, sl, jt) and ignore its result;
   [Statement$^{□}$ □ ForStatement$^{□}$] do
      Evaluate Validate[ForStatement$^{□}$](cxt, env, sl, jt) and ignore its result;
   [Statement$^{□}$ □ WithStatement$^{□}$] do
      Evaluate Validate[WithStatement$^{□}$](cxt, env, jt) and ignore its result;
   [Statement$^{□}$ □ ContinueStatement Semicolon$^{□}$] do
      Evaluate Validate[ContinueStatement](jt) and ignore its result;
   [Statement$^{□}$ □ BreakStatement Semicolon$^{□}$] do
      Evaluate Validate[BreakStatement](jt) and ignore its result;
   [Statement$^{□}$ □ ReturnStatement Semicolon$^{□}$] do
      Evaluate Validate[ReturnStatement](cxt, env) and ignore its result;
```

```
    [Statement$^{\omega}$ ⇒ ThrowStatement Semicolon$^{\omega}$] do
        Evaluate Validate[ThrowStatement](cxt, env) and ignore its result;
    [Statement$^{\omega}$ ⇒ TryStatement] do
        Evaluate Validate[TryStatement](cxt, env, jt) and ignore its result
end proc;

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

proc Validate[Substatement$^{\omega}$] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
    [Substatement$^{\omega}$ ⇒ EmptyStatement] do nothing;
    [Substatement$^{\omega}$ ⇒ Statement$^{\omega}$] do
        Evaluate Validate[Statement$^{\omega}$](cxt, env, sl, jt, false) and ignore its result;
    [Substatement$^{\omega}$ ⇒ SimpleVariableDefinition Semicolon$^{\omega}$] do
        Evaluate Validate[SimpleVariableDefinition](cxt, env) and ignore its result;
    [Substatement$^{\omega}$ ⇒ Attributes [no line break] { Substatements }] do
        Evaluate Validate[Attributes](cxt, env) and ignore its result;
        Evaluate Setup[Attributes]() and ignore its result;
        attr: ATTRIBUTE ← Eval[Attributes](env, compile);
        if attr ∉ BOOLEAN then
            throw a TypeError exception — attributes other than true and false may be used in a statement but not a
                    substatement
        end if;
        Enabled[Substatement$^{\omega}$] ← attr;
        if attr then Evaluate Validate[Substatements](cxt, env, jt) and ignore its result
        end if
end proc;

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

proc Validate[SubstatementsPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [SubstatementsPrefix ⇒ «empty»] do nothing;
    [SubstatementsPrefix$_0$ ⇒ SubstatementsPrefix$_1$ Substatement$^{full}$] do
        Evaluate Validate[SubstatementsPrefix$_1$](cxt, env, jt) and ignore its result;
        Evaluate Validate[Substatement$^{full}$](cxt, env, {}, jt) and ignore its result
end proc;
```

Setup

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

```
proc Setup[Substatement$^{\omega}$] ()
    [Substatement$^{\omega}$ ⇒ EmptyStatement] do nothing;
    [Substatement$^{\omega}$ ⇒ Statement$^{\omega}$] do Evaluate Setup[Statement$^{\omega}$]() and ignore its result;
    [Substatement$^{\omega}$ ⇒ SimpleVariableDefinition Semicolon$^{\omega}$] do
        Evaluate Setup[SimpleVariableDefinition]() and ignore its result;
    [$Substatement^{\omega}$ $\Rightarrow$ Attributes [no line break] { Substatements }] do
        if Enabled[$Substatement^{\omega}$] then
            Evaluate Setup[Substatements]() and ignore its result
        end if
end proc;
```

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

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

```
proc Setup[$Semicolon^{\omega}$] ()
    [$Semicolon^{\omega}$ $\Rightarrow$ ;] do nothing;
    [$Semicolon^{\omega}$ $\Rightarrow$ VirtualSemicolon] do nothing;
    [$Semicolon^{abbrev}$ $\Rightarrow$ «empty»] do nothing;
    [$Semicolon^{noShortIf}$ $\Rightarrow$ «empty»] do nothing
end proc;
```

Evaluation

```
proc Eval[$Statement^{\omega}$] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [$Statement^{\omega}$ $\Rightarrow$ ExpressionStatement $Semicolon^{\omega}$] do
        return Eval[ExpressionStatement](env);
    [$Statement^{\omega}$ $\Rightarrow$ SuperStatement $Semicolon^{\omega}$] do return Eval[SuperStatement](env);
    [$Statement^{\omega}$ $\Rightarrow$ Block] do return Eval[Block](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ $LabeledStatement^{\omega}$] do return Eval[$LabeledStatement^{\omega}$](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ $IfStatement^{\omega}$] do return Eval[$IfStatement^{\omega}$](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ SwitchStatement] do return Eval[SwitchStatement](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ DoStatement $Semicolon^{\omega}$] do return Eval[DoStatement](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ $WhileStatement^{\omega}$] do return Eval[$WhileStatement^{\omega}$](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ $ForStatement^{\omega}$] do return Eval[$ForStatement^{\omega}$](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ $WithStatement^{\omega}$] do return Eval[$WithStatement^{\omega}$](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ ContinueStatement $Semicolon^{\omega}$] do
        return Eval[ContinueStatement](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ BreakStatement $Semicolon^{\omega}$] do return Eval[BreakStatement](env, d);
    [$Statement^{\omega}$ $\Rightarrow$ ReturnStatement $Semicolon^{\omega}$] do return Eval[ReturnStatement](env);
    [$Statement^{\omega}$ $\Rightarrow$ ThrowStatement $Semicolon^{\omega}$] do return Eval[ThrowStatement](env);
    [$Statement^{\omega}$ $\Rightarrow$ TryStatement] do return Eval[TryStatement](env, d)
end proc;

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

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

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

12.1 Empty Statement

Syntax

EmptyStatement □ `;`

12.2 Expression Statement

Syntax

ExpressionStatement □ [lookahead□{**`function`**, **`{`**}] *ListExpression*^{allowIn}

Validation

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

Setup

Setup[*ExpressionStatement*] () propagates the call to Setup to nonterminals in the expansion of *ExpressionStatement*.

Evaluation

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

12.3 Super Statement

Syntax

SuperStatement □ **`super`** *Arguments*

Validation

```
proc Validate[SuperStatement → super Arguments] (cxt: CONTEXT, env: ENVIRONMENT)
   frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
   if frame = none or frame.kind ≠ constructorFunction then
      throw a SyntaxError exception — a super statement is meaningful only inside a constructor
   end if;
   Evaluate Validate[Arguments](cxt, env) and ignore its result;
   frame.callsSuperconstructor ← true
end proc;
```

Setup

Setup[*SuperStatement*] () propagates the call to Setup to nonterminals in the expansion of *SuperStatement*.

Evaluation

```
proc Eval[SuperStatement → super Arguments] (env: ENVIRONMENT): OBJECT
   frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
   note Validate already ensured that frame ≠ none and frame.kind = constructorFunction.
   args: OBJECT[] ← Eval[Arguments](env, run);
   if frame.superconstructorCalled = true then
      throw a ReferenceError exception — the superconstructor cannot be called twice
   end if;
   c: CLASS ← getEnclosingClass(env);
   this: OBJECTOPT ← frame.this;
   note this ∈ SIMPLEINSTANCE;
   Evaluate callInit(this, c.super, args, run) and ignore its result;
   frame.superconstructorCalled ← true;
   return this
end proc;
```

12.4 Block Statement

Syntax

Block → **{** *Directives* **}**

Validation

CompileFrame[*Block*]: LOCALFRAME;

Preinstantiate[*Block*]: BOOLEAN;

```
proc ValidateUsingFrame[Block → { Directives }]
      (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN, frame: FRAME)
   localCxt: CONTEXT ← new CONTEXT⟨⟨strict: cxt.strict, openNamespaces: cxt.openNamespaces⟩⟩
   Evaluate Validate[Directives](localCxt, [frame] ⊕ env, jt, preinst, none) and ignore its result
end proc;
```

```
proc Validate[Block ⇒ { Directives }] (cxt: Context, env: Environment, jt: JumpTargets, preinst: Boolean)
   compileFrame: LocalFrame ← new LocalFrame⟨⟨localBindings: {}⟩⟩;
   CompileFrame[Block] ← compileFrame;
   Preinstantiate[Block] ← preinst;
   Evaluate ValidateUsingFrame[Block](cxt, env, jt, preinst, compileFrame) and ignore its result
end proc;
```

Setup

Setup[*Block*] () propagates the call to Setup to nonterminals in the expansion of *Block*.

Evaluation

```
proc Eval[Block ⇒ { Directives }] (env: Environment, d: Object): Object
   compileFrame: LocalFrame ← CompileFrame[Block];
   runtimeFrame: LocalFrame;
   if Preinstantiate[Block] then runtimeFrame ← compileFrame
   else runtimeFrame ← instantiateLocalFrame(compileFrame, env)
   end if;
   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;
```

12.5 Labeled Statements

Syntax

LabeledStatement$^{\omega}$ ⇒ *Identifier* : *Substatement*$^{\omega}$

Validation

```
proc Validate[LabeledStatement^ω ⇒ Identifier : Substatement^ω]
      (cxt: Context, env: Environment, sl: Label{}, jt: JumpTargets)
   name: String ← Name[Identifier];
   if name ∈ jt.breakTargets then
      throw a SyntaxError exception — nesting labeled statements with the same label is not permitted
   end if;
   jt2: JumpTargets ← JumpTargets⟨⟨breakTargets: jt.breakTargets ∪ {name},
         continueTargets: jt.continueTargets⟩⟩;
   Evaluate Validate[Substatement^ω](cxt, env, sl ∪ {name}, jt2) and ignore its result
end proc;
```

Setup

```
proc Setup[LabeledStatement^ω ⇒ Identifier : Substatement^ω] ()
   Evaluate Setup[Substatement^ω]() and ignore its result
end proc;
```

Evaluation

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

12.6 If Statement

Syntax

```
IfStatement^abbrev □
      if ParenListExpression Substatement^abbrev
   |  if ParenListExpression Substatement^noShortIf else Substatement^abbrev

IfStatement^full □
      if ParenListExpression Substatement^full
   |  if ParenListExpression Substatement^noShortIf else Substatement^full

IfStatement^noShortIf □ if ParenListExpression Substatement^noShortIf else Substatement^noShortIf
```

Validation

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

Setup

Setup[*IfStatement*□] () propagates the call to Setup to nonterminals in the expansion of *IfStatement*□.

Evaluation

```
proc Eval[IfStatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [IfStatement^abbrev □ if ParenListExpression Substatement^abbrev] do
        o: OBJECT □ readReference(Eval[ParenListExpression](env, run), run);
        if objectToBoolean(o) then return Eval[Substatement^abbrev](env, d)
        else return d
        end if;
    [IfStatement^full ⇒ if ParenListExpression Substatement^full] do
       o: OBJECT ← readReference(Eval[ParenListExpression](env, run), run);
       if objectToBoolean(o) then return Eval[Substatement^full](env, d)
       else return d
       end if;
    [IfStatement^ω ⇒ if ParenListExpression Substatement^noShortIf_1 else Substatement^ω_2] do
       o: OBJECT ← readReference(Eval[ParenListExpression](env, run), run);
       if objectToBoolean(o) then return Eval[Substatement^noShortIf_1](env, d)
       else return Eval[Substatement^ω_2](env, d)
       end if
 end proc;
```

12.7 Switch Statement

Semantics

```
 tuple SWITCHKEY
    key: OBJECT
 end tuple;

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

Syntax

```
 SwitchStatement ⇒ switch ParenListExpression { CaseElements }

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

 CaseElementsPrefix ⇒
      «empty»
   |  CaseElementsPrefix CaseElement^full

 CaseElement^ω ⇒
      Directive^ω
   |  CaseLabel

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

Validation

```
 CompileFrame[SwitchStatement]: LOCALFRAME;
```

```
proc Validate[SwitchStatement → switch ParenListExpression { CaseElements }]
      (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   if NDefaults[CaseElements] > 1 then
      throw a SyntaxError exception — a case statement may have at most one default clause
   end if;
   Evaluate Validate[ParenListExpression](cxt, env) and ignore its result;
   jt2: JUMPTARGETS ← JUMPTARGETS⟨⟨breakTargets: jt.breakTargets ∪ {default},
         continueTargets: jt.continueTargets⟩⟩;
   compileFrame: LOCALFRAME ← new LOCALFRAME⟨⟨localBindings: {}⟩⟩;
   CompileFrame[SwitchStatement] ← compileFrame;
   localCxt: CONTEXT ← new CONTEXT⟨⟨strict: cxt.strict, openNamespaces: cxt.openNamespaces⟩⟩;
   Evaluate Validate[CaseElements](localCxt, [compileFrame] ⊕ env, jt2) and ignore its result
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 nonterminals 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 nonterminals in the expansion of CaseElementsPrefix.

```
NDefaults[CaseElement^ω]: INTEGER;
   NDefaults[CaseElement^ω → Directive^ω] = 0;
   NDefaults[CaseElement^ω → CaseLabel] = NDefaults[CaseLabel];
```

```
proc Validate[CaseElement^ω] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
   [CaseElement^ω → Directive^ω] do
      Evaluate Validate[Directive^ω](cxt, env, jt, false, none) and ignore its result;
   [CaseElement^ω → CaseLabel] do
      Evaluate Validate[CaseLabel](cxt, env, jt) and ignore its result
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
      Evaluate Validate[ListExpression^allowIn](cxt, env) and ignore its result;
   [CaseLabel → default :] do nothing
end proc;
```

Setup

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

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

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

Setup[*CaseElement*^{ω}] () propagates the call to Setup to nonterminals in the expansion of *CaseElement*^{ω}.

Setup[*CaseLabel*] () propagates the call to Setup to nonterminals 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^abbrev] do
        guard2: SWITCHGUARD ← Eval[CaseLabel](env, guard, d);
        guard3: SWITCHGUARD ← Eval[CaseElementsPrefix](env, guard2, d);
        return Eval[CaseElement^abbrev](env, guard3, d)
end proc;

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

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

```
proc Eval[CaseLabel] (env: ENVIRONMENT, guard: SWITCHGUARD, d: OBJECT): SWITCHGUARD
   [CaseLabel □ case ListExpression^allowIn :] do
      case guard of
         {default} □ OBJECT do return guard;
         SWITCHKEY do
            label: OBJECT □ readReference(Eval[ListExpression^allowIn](env, run), run);
            if isStrictlyEqual(guard.key, label, run) then return d
            else return guard
            end if
      end case;
   [CaseLabel □ default :] do
      case guard of
         SWITCHKEY □ OBJECT do return guard;
         {default} do return d
      end case
end proc;
```

12.8 Do-While Statement

Syntax

```
DoStatement □ do Substatement^abbrev while ParenListExpression
```

Validation

```
Labels[DoStatement]: LABEL{};

proc Validate[DoStatement □ do Substatement^abbrev while ParenListExpression]
      (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
   continueLabels: LABEL{} □ sl □ {default};
   Labels[DoStatement] □ continueLabels;
   jt2: JUMPTARGETS □ JUMPTARGETS□breakTargets: jt.breakTargets □ {default},
         continueTargets: jt.continueTargets □ continueLabels□;
   Evaluate Validate[Substatement^abbrev](cxt, env, {}, jt2) and ignore its result;
   Evaluate Validate[ParenListExpression](cxt, env) and ignore its result
end proc;
```

Setup

Setup[*DoStatement*] () propagates the call to Setup to nonterminals 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 objectToBoolean(o) 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;
```

12.9 While Statement

Syntax

$WhileStatement^{□}$ □ **while** ParenListExpression $Substatement^{□}$

Validation

```
Labels[$WhileStatement^{□}$]: LABEL{};

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

Setup

Setup[$WhileStatement^{□}$] () propagates the call to Setup to nonterminals in the expansion of $WhileStatement^{□}$.

Evaluation

```
proc Eval[WhileStatement□ □ while ParenListExpression Substatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
  try
    d1: OBJECT □ d;
    while objectToBoolean(readReference(Eval[ParenListExpression](env, 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;
```

12.10 For Statements

Syntax

ForStatement□ □
  for (*ForInitializer* **;** *OptionalExpression* **;** *OptionalExpression* **)** *Substatement*□
  | **for (** *ForInBinding* **in** *ListExpression*$^{\text{allowIn}}$ **)** *Substatement*□

ForInitializer □
  «empty»
  | *ListExpression*$^{\text{noIn}}$
  | *VariableDefinition*$^{\text{noIn}}$
  | *Attributes* [no line break] *VariableDefinition*$^{\text{noIn}}$

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

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

Validation

Labels[*ForStatement*□]: LABEL{};

CompileLocalFrame[*ForStatement*□]: LOCALFRAME;

```
proc Validate[ForStatement$^{\omega}$] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
   [ForStatement$^{\omega}$ ⇒ for ( ForInitializer ; OptionalExpression$_1$ ; OptionalExpression$_2$ ) Substatement$^{\omega}$] do
      continueLabels: LABEL{} ← sl ∪ {default};
      Labels[ForStatement$^{\omega}$] ← continueLabels;
      jt2: JUMPTARGETS ← JUMPTARGETS⟨⟨breakTargets: jt.breakTargets ∪ {default},
            continueTargets: jt.continueTargets ∪ continueLabels⟩⟩;
      compileLocalFrame: LOCALFRAME ← new LOCALFRAME⟨⟨localBindings: {}⟩⟩;
      CompileLocalFrame[ForStatement$^{\omega}$] ← compileLocalFrame;
      compileEnv: ENVIRONMENT ← [compileLocalFrame] ⊕ env;
      Evaluate Validate[ForInitializer](cxt, compileEnv) and ignore its result;
      Evaluate Validate[OptionalExpression$_1$](cxt, compileEnv) and ignore its result;
      Evaluate Validate[OptionalExpression$_2$](cxt, compileEnv) and ignore its result;
      Evaluate Validate[Substatement$^{\omega}$](cxt, compileEnv, {}, jt2) and ignore its result;
   [ForStatement$^{\omega}$ ⇒ for ( ForInBinding in ListExpression$^{allowIn}$ ) Substatement$^{\omega}$] do
      continueLabels: LABEL{} ← sl ∪ {default};
      Labels[ForStatement$^{\omega}$] ← continueLabels;
      jt2: JUMPTARGETS ← JUMPTARGETS⟨⟨breakTargets: jt.breakTargets ∪ {default},
            continueTargets: jt.continueTargets ∪ continueLabels⟩⟩;
      Evaluate Validate[ListExpression$^{allowIn}$](cxt, env) and ignore its result;
      compileLocalFrame: LOCALFRAME ← new LOCALFRAME⟨⟨localBindings: {}⟩⟩;
      CompileLocalFrame[ForStatement$^{\omega}$] ← compileLocalFrame;
      compileEnv: ENVIRONMENT ← [compileLocalFrame] ⊕ env;
      Evaluate Validate[ForInBinding](cxt, compileEnv) and ignore its result;
      Evaluate Validate[Substatement$^{\omega}$](cxt, compileEnv, {}, jt2) and ignore its result
end proc;

Enabled[ForInitializer]: BOOLEAN;

proc Validate[ForInitializer] (cxt: CONTEXT, env: ENVIRONMENT)
   [ForInitializer ⇒ «empty»] do nothing;
   [ForInitializer ⇒ ListExpression$^{noIn}$] do
      Evaluate Validate[ListExpression$^{noIn}$](cxt, env) and ignore its result;
   [ForInitializer ⇒ VariableDefinition$^{noIn}$] do
      Evaluate Validate[VariableDefinition$^{noIn}$](cxt, env, none) and ignore its result;
   [ForInitializer ⇒ Attributes [no line break] VariableDefinition$^{noIn}$] do
      Evaluate Validate[Attributes](cxt, env) and ignore its result;
      Evaluate Setup[Attributes]() and ignore its result;
      attr: ATTRIBUTE ← Eval[Attributes](env, compile);
      Enabled[ForInitializer] ← attr ≠ false;
      if attr ≠ false then
         Evaluate Validate[VariableDefinition$^{noIn}$](cxt, env, attr) and ignore its result
      end if
end proc;

proc Validate[ForInBinding] (cxt: CONTEXT, env: ENVIRONMENT)
   [ForInBinding ⇒ PostfixExpression] do
      Evaluate Validate[PostfixExpression](cxt, env) and ignore its result;
   [ForInBinding ⇒ VariableDefinitionKind VariableBinding$^{noIn}$] do
      Evaluate Validate[VariableBinding$^{noIn}$](cxt, env, none, Immutable[VariableDefinitionKind], true) and ignore its
            result;
```

```
    [ForInBinding ⇒ Attributes [no line break] VariableDefinitionKind VariableBinding^noIn] do
        Evaluate Validate[Attributes](cxt, env) and ignore its result;
        Evaluate Setup[Attributes]() and ignore its result;
        attr: ATTRIBUTE ← Eval[Attributes](env, compile);
        if attr = false then
            throw an AttributeError exception — the false attribute canot be applied to a for-in variable definition
        end if;
        Evaluate Validate[VariableBinding^noIn](cxt, env, attr, Immutable[VariableDefinitionKind], true) and ignore its
                result
end proc;
```

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

Setup

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

```
proc Setup[ForInitializer] ()
    [ForInitializer ⇒ «empty»] do nothing;
    [ForInitializer ⇒ ListExpression^noIn] do
        Evaluate Setup[ListExpression^noIn]() and ignore its result;
    [ForInitializer ⇒ VariableDefinition^noIn] do
        Evaluate Setup[VariableDefinition^noIn]() and ignore its result;
    [ForInitializer ⇒ Attributes [no line break] VariableDefinition^noIn] do
        if Enabled[ForInitializer] then
            Evaluate Setup[VariableDefinition^noIn]() and ignore its result
        end if
end proc;

proc Setup[ForInBinding] ()
    [ForInBinding ⇒ PostfixExpression] do
        Evaluate Setup[PostfixExpression]() and ignore its result;
    [ForInBinding ⇒ VariableDefinitionKind VariableBinding^noIn] do
        Evaluate Setup[VariableBinding^noIn]() and ignore its result;
    [ForInBinding ⇒ Attributes [no line break] VariableDefinitionKind VariableBinding^noIn] do
        Evaluate Setup[VariableBinding^noIn]() and ignore its result
end proc;
```

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

Evaluation

```
proc Eval[ForStatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [ForStatement□ □ for ( ForInitializer ; OptionalExpression$_1$ ; OptionalExpression$_2$ ) Substatement□] do
    runtimeLocalFrame: LOCALFRAME □ instantiateLocalFrame(CompileLocalFrame[ForStatement□], env);
    runtimeEnv: ENVIRONMENT □ [runtimeLocalFrame] ⊕ env;
    try
      Evaluate Eval[ForInitializer](runtimeEnv) and ignore its result;
      d1: OBJECT □ d;
      while objectToBoolean(readReference(Eval[OptionalExpression$_1$](runtimeEnv, run), run)) do
        try d1 □ Eval[Substatement□](runtimeEnv, d1)
        catch x: SEMANTICEXCEPTION do
          if x □ CONTINUE and x.label □ Labels[ForStatement□] then
            d1 □ x.value
          else throw x
          end if
        end try;
        Evaluate readReference(Eval[OptionalExpression$_2$](runtimeEnv, run), run) and ignore its result
      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□ □ for ( ForInBinding in ListExpressionallowIn ) Substatement□] do
       try
          o: OBJECT □ readReference(Eval[ListExpressionallowIn](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□], env);
             runtimeEnv: ENVIRONMENT □ [runtimeLocalFrame] ⊕ env;
             index: OBJECT □ any element of remainingIndices;
             remainingIndices □ remainingIndices – {index};
             Evaluate WriteBinding[ForInBinding](runtimeEnv, index) and ignore its result;
             try d1 □ Eval[Substatement□](runtimeEnv, d1)
             catch x: SEMANTICEXCEPTION do
                if x □ CONTINUE and x.label □ Labels[ForStatement□] 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[ForInitializer] (env: ENVIRONMENT)
    [ForInitializer □ «empty»] do nothing;
    [ForInitializer □ ListExpressionnoIn] do
       Evaluate readReference(Eval[ListExpressionnoIn](env, run), run) and ignore its result;
    [ForInitializer □ VariableDefinitionnoIn] do
       Evaluate Eval[VariableDefinitionnoIn](env, undefined) and ignore its result;
    [ForInitializer □ Attributes [no line break] VariableDefinitionnoIn] do
       if Enabled[ForInitializer] then
          Evaluate Eval[VariableDefinitionnoIn](env, undefined) and ignore its result
       end if
 end proc;

 proc WriteBinding[ForInBinding] (env: ENVIRONMENT, newValue: OBJECT)
    [ForInBinding □ PostfixExpression] do
       r: OBJORREF □ Eval[PostfixExpression](env, run);
       Evaluate writeReference(r, newValue, run) and ignore its result;
    [ForInBinding □ VariableDefinitionKind VariableBindingnoIn] do
       Evaluate WriteBinding[VariableBindingnoIn](env, newValue) and ignore its result;
    [ForInBinding □ Attributes [no line break] VariableDefinitionKind VariableBinding$^{noIn}$] do
      Evaluate WriteBinding[VariableBinding$^{noIn}$](env, newValue) and ignore its result
  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;
```

12.11 With Statement

Syntax

```
WithStatement$^{□}$ □ with ParenListExpression Substatement$^{□}$
```

Validation

```
  CompileLocalFrame[WithStatement$^{□}$]: LOCALFRAME;

  proc Validate[WithStatement$^{□}$ □ with ParenListExpression Substatement$^{□}$]
      (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    Evaluate Validate[ParenListExpression](cxt, env) and ignore its result;
    compileWithFrame: WITHFRAME □ new WITHFRAME□value: none□
    compileLocalFrame: LOCALFRAME □ new LOCALFRAME□localBindings: {}□
    CompileLocalFrame[WithStatement$^{□}$] □ compileLocalFrame;
    compileEnv: ENVIRONMENT □ [compileLocalFrame] ⊕ [compileWithFrame] ⊕ env;
    Evaluate Validate[Substatement$^{□}$](cxt, compileEnv, {}, jt) and ignore its result
  end proc;
```

Setup

Setup[WithStatement$^{□}$] () propagates the call to Setup to nonterminals in the expansion of WithStatement$^{□}$.

Evaluation

```
  proc Eval[WithStatement$^{□}$ □ with ParenListExpression Substatement$^{□}$] (env: ENVIRONMENT, d: OBJECT): OBJECT
    value: OBJECT □ readReference(Eval[ParenListExpression](env, run), run);
    runtimeWithFrame: WITHFRAME □ new WITHFRAME□value: value□
    runtimeLocalFrame: LOCALFRAME □
        instantiateLocalFrame(CompileLocalFrame[WithStatement$^{□}$], [runtimeWithFrame] ⊕ env);
    runtimeEnv: ENVIRONMENT □ [runtimeLocalFrame] ⊕ [runtimeWithFrame] ⊕ env;
    return Eval[Substatement$^{□}$](runtimeEnv, d)
  end proc;
```

12.12 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 a SyntaxError exception — there is no enclosing statement to which to continue
        end if;
    [ContinueStatement □ continue [no line break] Identifier] do
        if Name[Identifier] □ jt.continueTargets then
            throw a SyntaxError exception — there is no enclosing labeled statement to which to continue
        end if
end proc;

proc Validate[BreakStatement] (jt: JUMPTARGETS)
    [BreakStatement □ break] do
        if default □ jt.breakTargets then
            throw a SyntaxError exception — there is no enclosing statement to which to break
        end if;
    [BreakStatement □ break [no line break] Identifier] do
        if Name[Identifier] □ jt.breakTargets then
            throw a SyntaxError exception — there is no enclosing labeled statement to which to break
        end if
end proc;
```

Setup

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

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

Evaluation

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

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

12.13 Return Statement

Syntax

```
ReturnStatement □
    return
  | return [no line break] ListExpression^allowIn
```

Validation

```
proc Validate[ReturnStatement] (cxt: Context, env: Environment)
   [ReturnStatement □ return] do
      if getEnclosingParameterFrame(env) = none then
         throw a SyntaxError exception — a return statement must be located inside a function
      end if;
   [ReturnStatement □ return [no line break] ListExpression^allowIn] do
      frame: ParameterFrameOpt □ getEnclosingParameterFrame(env);
      if frame = none then
         throw a SyntaxError exception — a return statement must be located inside a function
      end if;
      if cannotReturnValue(frame) then
         throw a SyntaxError exception — a return statement inside a setter or constructor cannot return a value
      end if;
      Evaluate Validate[ListExpression^allowIn](cxt, env) and ignore its result
end proc;
```

Setup

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

Evaluation

```
proc Eval[ReturnStatement] (env: Environment): Object
   [ReturnStatement □ return] do throw Return□value: undefined□
   [ReturnStatement □ return [no line break] ListExpression^allowIn] do
      a: Object □ readReference(Eval[ListExpression^allowIn](env, run), run);
      throw Return□value: a□
end proc;
```

cannotReturnValue(*frame*) returns **true** if the function represented by *frame* cannot return a value because it is a setter or constructor.

```
proc cannotReturnValue(frame: ParameterFrame): Boolean
   return frame.kind = constructorFunction or frame.handling = set
end proc;
```

12.14 Throw Statement

Syntax

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

Validation

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

Setup

Setup[*ThrowStatement*] () propagates the call to Setup to nonterminals in the expansion of *ThrowStatement*.

Evaluation

```
proc Eval[ThrowStatement □ throw [no line break] ListExpression^allowIn] (env: ENVIRONMENT): OBJECT
    a: OBJECT □ readReference(Eval[ListExpression^allowIn](env, run), run);
    throw a
end proc;
```

12.15 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
        Evaluate Validate[Block](cxt, env, jt, false) and ignore its result;
        Evaluate Validate[CatchClauses](cxt, env, jt) and ignore its result;
    [TryStatement □ try Block_1 CatchClausesOpt finally Block_2] do
        Evaluate Validate[Block_1](cxt, env, jt, false) and ignore its result;
        Evaluate Validate[CatchClausesOpt](cxt, env, jt) and ignore its result;
        Evaluate Validate[Block_2](cxt, env, jt, false) and ignore its result
end proc;
```

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

Validate[*CatchClauses*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS) propagates the call to Validate to nonterminals 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: {}⟩⟩;
   compileEnv: ENVIRONMENT ← [compileFrame] ⊕ env;
   CompileFrame[CatchClause] ← compileFrame;
   CompileEnv[CatchClause] ← compileEnv;
   Evaluate Validate[Parameter](cxt, compileEnv, compileFrame) and ignore its result;
   Evaluate Validate[Block](cxt, compileEnv, jt, false) and ignore its result
end proc;
```

Setup

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

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

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

```
proc Setup[CatchClause ⇒ catch ( Parameter ) Block] ()
   Evaluate Setup[Parameter](CompileEnv[CatchClause], CompileFrame[CatchClause], none) and ignore its result;
   Evaluate Setup[Block]() and ignore its result
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 ∈ CONTROLTRANSFER then throw x
         else
            r: OBJECT ∪ {reject} ← Eval[CatchClauses](env, x);
            if r ≠ reject then return r else throw x end if
         end if
      end try;
```

```
    [TryStatement □ try Block_1 CatchClausesOpt finally Block_2] do
      result: OBJECTOPT □ none;
      exception: SEMANTICEXCEPTION □ {none} □ none;
      try result □ Eval[Block_1](env, d)
      catch x: SEMANTICEXCEPTION do exception □ x
      end try;
      note At this point exactly one of result and exception has a non-none value.
      if exception □ OBJECT then
        try
          r: OBJECT □ {reject} □ Eval[CatchClausesOpt](env, exception);
          if $r \neq$ reject then
            note The exception has been handled, so clear it.
            result □ r;
            exception □ none
          end if
        catch x: SEMANTICEXCEPTION do
          note The catch clause threw another exception or CONTROLTRANSFER x, so replace the original exception
              with x.
          exception □ x
        end try
      end if;
      note The finally clause is executed even if the original block exited due to a CONTROLTRANSFER (break,
          continue, or return).
      note The finally clause is not inside a try-catch semantic statement, so if it throws another exception or
          CONTROLTRANSFER, then the original exception or CONTROLTRANSFER exception is dropped.
      Evaluate Eval[Block_2](env, undefined) and ignore its result;
      note At this point exactly one of result and exception has a non-none value.
      if $exception \neq$ none then throw exception else return result end if
  end proc;

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

  proc Eval[CatchClauses] (env: ENVIRONMENT, exception: OBJECT): OBJECT □ {reject}
    [CatchClauses □ CatchClause] do return Eval[CatchClause](env, exception);
    [CatchClauses_0 □ CatchClauses_1 CatchClause] do
      r: OBJECT □ {reject} □ Eval[CatchClauses_1](env, exception);
      if $r \neq$ 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: SINGLETONPROPERTYOPT ← findLocalSingletonProperty(runtimeFrame, {qname}, write);
   note Validate created one local variable with the name in qname, so v ∈ VARIABLE.
   if is(exception, v.type) then
      Evaluate writeSingletonProperty(v, exception, run) and ignore its result;
      return Eval[Block](runtimeEnv, undefined)
   else return reject
   end if
end proc;
```

13 Directives

Syntax

```
Directive^ω →
      EmptyStatement
   |  Statement^ω
   |  AnnotatableDirective^ω
   |  Attributes [no line break] AnnotatableDirective^ω
   |  Attributes [no line break] { Directives }
   |  Pragma Semicolon^ω

AnnotatableDirective^ω →
      VariableDefinition^allowIn Semicolon^ω
   |  FunctionDefinition
   |  ClassDefinition
   |  NamespaceDefinition Semicolon^ω
   |  ImportDirective Semicolon^ω
   |  UseDirective Semicolon^ω

Directives →
      «empty»
   |  DirectivesPrefix Directive^abbrev

DirectivesPrefix →
      «empty»
   |  DirectivesPrefix Directive^full
```

Validation

```
Enabled[Directive^ω]: BOOLEAN;

proc Validate[Directive^ω] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN,
      attr: ATTRIBUTEOPTNOTFALSE)
   [Directive^ω → EmptyStatement] do nothing;
   [Directive^ω → Statement^ω] do
      if attr ∉ {none, true} then
         throw an AttributeError exception — an ordinary statement only permits the attributes true and false
      end if;
      Evaluate Validate[Statement^ω](cxt, env, {}, jt, preinst) and ignore its result;
```

```
    [Directive□ □  AnnotatableDirective□] do
       Evaluate Validate[AnnotatableDirective□](cxt, env, preinst, attr) and ignore its result;
    [Directive□ □  Attributes [no line break] AnnotatableDirective□] do
       Evaluate Validate[Attributes](cxt, env) and ignore its result;
       Evaluate Setup[Attributes]() and ignore its result;
       attr2: ATTRIBUTE □  Eval[Attributes](env, compile);
       attr3: ATTRIBUTE □  combineAttributes(attr, attr2);
       if attr3 = false then Enabled[Directive□] □  false
       else
          Enabled[Directive□] □  true;
          Evaluate Validate[AnnotatableDirective□](cxt, env, preinst, attr3) and ignore its result
       end if;
    [Directive□ □  Attributes [no line break] { Directives }] do
       Evaluate Validate[Attributes](cxt, env) and ignore its result;
       Evaluate Setup[Attributes]() and ignore its result;
       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□
          Evaluate Validate[Directives](localCxt, env, jt, preinst, attr3) and ignore its result
       end if;
    [Directive□ □  Pragma Semicolon□] do
       if attr □ {none, true} then Evaluate Validate[Pragma](cxt) and ignore its result
       else
          throw an AttributeError exception — a pragma directive only permits the attributes true and false
       end if
 end proc;

 proc Validate[AnnotatableDirective□]
       (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
    [AnnotatableDirective□ □  VariableDefinition^allowIn Semicolon□] do
       Evaluate Validate[VariableDefinition^allowIn](cxt, env, attr) and ignore its result;
    [AnnotatableDirective□ □  FunctionDefinition] do
       Evaluate Validate[FunctionDefinition](cxt, env, preinst, attr) and ignore its result;
    [AnnotatableDirective□ □  ClassDefinition] do
       Evaluate Validate[ClassDefinition](cxt, env, preinst, attr) and ignore its result;
    [AnnotatableDirective□ □  NamespaceDefinition Semicolon□] do
       Evaluate Validate[NamespaceDefinition](cxt, env, preinst, attr) and ignore its result;
    [AnnotatableDirective□ □  ImportDirective Semicolon□] do
       Evaluate Validate[ImportDirective](cxt, env, preinst, attr) and ignore its result;
    [AnnotatableDirective□ □  UseDirective Semicolon□] do
       if attr □ {none, true} then
          Evaluate Validate[UseDirective](cxt, env) and ignore its result
       else
          throw an AttributeError exception — a use directive only permits the attributes true and false
       end if
 end proc;
```

Validate[*Directives*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS, *preinst*: BOOLEAN, *attr*: ATTRIBUTEOPTNOTFALSE) propagates the call to Validate to nonterminals in the expansion of *Directives*.

Validate[*DirectivesPrefix*] (*cxt*: CONTEXT, *env*: ENVIRONMENT, *jt*: JUMPTARGETS, *preinst*: BOOLEAN, *attr*: ATTRIBUTEOPTNOTFALSE) propagates the call to Validate to nonterminals in the expansion of *DirectivesPrefix*.

Setup

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

```
proc Setup[AnnotatableDirective□] ()
  [AnnotatableDirective□ □ VariableDefinition^allowIn Semicolon□] do
    Evaluate Setup[VariableDefinition^allowIn]() and ignore its result;
  [AnnotatableDirective□ □ FunctionDefinition] do
    Evaluate Setup[FunctionDefinition]() and ignore its result;
  [AnnotatableDirective□ □ ClassDefinition] do
    Evaluate Setup[ClassDefinition]() and ignore its result;
  [AnnotatableDirective□ □ NamespaceDefinition Semicolon□] do nothing;
  [AnnotatableDirective□ □ ImportDirective Semicolon□] do nothing;
  [AnnotatableDirective□ □ UseDirective Semicolon□] do nothing
end proc;
```

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

Setup[*DirectivesPrefix*] () propagates the call to Setup to nonterminals 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□ □ Pragma Semicolon□] do return d
end proc;

proc Eval[AnnotatableDirective□] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [AnnotatableDirective□ □ VariableDefinition^allowIn Semicolon□] do
        return Eval[VariableDefinition^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 return d;
    [AnnotatableDirective□ □ UseDirective Semicolon□] do return d
end proc;

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

proc Eval[DirectivesPrefix] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [DirectivesPrefix □ «empty»] do return d;
    [DirectivesPrefix_0 □ DirectivesPrefix_1 Directive^full] do
        o: OBJECT □ Eval[DirectivesPrefix_1](env, d);
        return Eval[Directive^full](env, o)
end proc;
```

13.1 Attributes

Syntax

```
Attributes □
      Attribute
   |  AttributeCombination

AttributeCombination □ Attribute [no line break] Attributes

Attribute □
      AttributeExpression
   |  true
   |  false
   |  ReservedNamespace
```

Validation

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

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

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

Setup

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

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

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

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);
      return objectToAttribute(a, phase);
   [Attribute □ true] do return true;
   [Attribute □ false] do return false;
   [Attribute □ ReservedNamespace] do return Eval[ReservedNamespace](env, phase)
end proc;
```

13.2 Use Directive

Syntax

UseDirective □ **use namespace** *ParenListExpression*

Validation

```
proc Validate[UseDirective □ use namespace ParenListExpression] (cxt: CONTEXT, env: ENVIRONMENT)
   Evaluate Validate[ParenListExpression](cxt, env) and ignore its result;
   Evaluate Setup[ParenListExpression]() and ignore its result;
   values: OBJECT[] □ EvalAsList[ParenListExpression](env, compile);
   namespaces: NAMESPACE{} □ {};
   for each v □ values do
      if v □ NAMESPACE then throw a TypeError exception end if;
      namespaces □ namespaces □ {v}
   end for each;
   cxt.openNamespaces □ cxt.openNamespaces □ namespaces
end proc;
```

13.3 Import Directive

Syntax

```
ImportDirective □
     import PackageName
   | import Identifier = PackageName
```

Validation

```
proc Validate[ImportDirective] (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
   [ImportDirective □ import PackageName] do
      if not preinst then
         throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
      end if;
      frame: FRAME □ env[0];
      if frame □ PACKAGE then
         throw a SyntaxError exception — a package may be imported only into a package scope
      end if;
      if attr □ {none, true} then
         throw an AttributeError exception — an unnamed import directive only permits the attributes true and
               false
      end if;
      pkgName: STRING □ Name[PackageName];
      pkg: PACKAGE □ locatePackage(pkgName);
      Evaluate importPackageInto(pkg, frame) and ignore its result;
   [ImportDirective □ import Identifier = PackageName] do
      if not preinst then
         throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
      end if;
      frame: FRAME □ env[0];
      if frame □ PACKAGE then
         throw a SyntaxError exception — a package may be imported only into a package scope
      end if;
      a: COMPOUNDATTRIBUTE □ toCompoundAttribute(attr);
      if a.dynamic then
         throw an AttributeError exception — a package definition cannot have the dynamic attribute
      end if;
      if a.prototype then
         throw an AttributeError exception — a package definition cannot have the prototype attribute
      end if;
      pkgName: STRING □ Name[PackageName];
      pkg: PACKAGE □ locatePackage(pkgName);
      v: VARIABLE □ new VARIABLE□type: Package, value: pkg, immutable: true, setup: none, initializer: none□
      Evaluate defineSingletonProperty(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite,
            v) and ignore its result;
      Evaluate importPackageInto(pkg, frame) and ignore its result
end proc;
```

```
proc locatePackage(name: STRING): PACKAGE
  Look for a package bound to name in the implementation's list of available packages. If one is found, let pkg: PACKAGE
  be that package; otherwise, throw an implementation-defined error.
  initialize: (() → ()) ∪ {none, busy} ← pkg.initialize;
  case initialize of
    {none} do nothing;
    {busy} do throw an UninitializedError exception — circular package dependency;
    () → () do
      Evaluate initialize() and ignore its result;
      note pkg.initialize = none;
  end case;
  return pkg
end proc;

proc importPackageInto(source: PACKAGE, destination: PACKAGE)
  for each b ∈ source.localBindings do
    if not (b.explicit or b.content = forbidden or (some d ∈ destination.localBindings satisfies
        b.qname = d.qname and accessesOverlap(b.accesses, d.accesses))) then
      destination.localBindings ← destination.localBindings ∪ {b}
    end if
  end for each
end proc;
```

13.4 Pragma

Syntax

```
Pragma ⇒ use PragmaItems

PragmaItems ⇒
    PragmaItem
  | PragmaItems , PragmaItem

PragmaItem ⇒
    PragmaExpr
  | PragmaExpr ?

PragmaExpr ⇒
    Identifier
  | Identifier ( PragmaArgument )

PragmaArgument ⇒
    true
  | false
  | Number
  | - Number
  | - NegatedMinLong
  | String
```

Validation

Validate[*Pragma*] (*cxt*: CONTEXT) propagates the call to Validate to nonterminals in the expansion of *Pragma*.

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

```
proc Validate[PragmaItem] (cxt: CONTEXT)
   [PragmaItem □ PragmaExpr] do
      Evaluate Validate[PragmaExpr](cxt, false) and ignore its result;
   [PragmaItem □ PragmaExpr ?] do
      Evaluate Validate[PragmaExpr](cxt, true) and ignore its result
end proc;

proc Validate[PragmaExpr] (cxt: CONTEXT, optional: BOOLEAN)
   [PragmaExpr □ Identifier] do
      Evaluate processPragma(cxt, Name[Identifier], undefined, optional) and ignore its result;
   [PragmaExpr □ Identifier ( PragmaArgument )] do
      arg: OBJECT □ Value[PragmaArgument];
      Evaluate processPragma(cxt, Name[Identifier], arg, optional) and ignore its result
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] = $(–2^{63})_{long}$;
   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, $4_{f64}$} then return end if;
      if value □ {$1_{f64}$, $2_{f64}$, $3_{f64}$} 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 a SyntaxError exception end if
end proc;
```

14 Definitions

14.1 Variable Definition

Syntax

```
VariableDefinition□ □ VariableDefinitionKind VariableBindingList□

VariableDefinitionKind □
     var
   | const
```

```
VariableBindingList□ □
     VariableBinding□
  |  VariableBindingList□ , VariableBinding□

VariableBinding□ □  TypedIdentifier□ VariableInitialisation□

VariableInitialisation□ □
     «empty»
  |  = VariableInitializer□

VariableInitializer□ □
     AssignmentExpression□
  |  AttributeCombination

TypedIdentifier□ □
     Identifier
  |  Identifier : TypeExpression□
```

Validation

```
proc Validate[VariableDefinition□ □  VariableDefinitionKind VariableBindingList□]
      (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE)
   Evaluate Validate[VariableBindingList□](cxt, env, attr, Immutable[VariableDefinitionKind], false) and ignore its
         result
end proc;

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

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

CompileEnv[VariableBinding□]: ENVIRONMENT;

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

OverriddenVar[VariableBinding□]: INSTANCEVARIABLEOPT;

Multiname[VariableBinding□]: MULTINAME;
```

```
proc Validate[VariableBinding^β ⇒ TypedIdentifier^β VariableInitialisation^β] (cxt: CONTEXT, env: ENVIRONMENT,
      attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN, noInitializer: BOOLEAN)
   Evaluate Validate[TypedIdentifier^β](cxt, env) and ignore its result;
   Evaluate Validate[VariableInitialisation^β](cxt, env) and ignore its result;
   CompileEnv[VariableBinding^β] ← env;
   name: STRING ← Name[TypedIdentifier^β];
   if not cxt.strict and getRegionalFrame(env) ∈ PACKAGE ∪ PARAMETERFRAME and not immutable and
         attr = none and Plain[TypedIdentifier^β] then
      qname: QUALIFIEDNAME ← public::name;
      Multiname[VariableBinding^β] ← {qname};
      CompileVar[VariableBinding^β] ← defineHoistedVar(env, name, undefined)
   else
      a: COMPOUNDATTRIBUTE ← toCompoundAttribute(attr);
      if a.dynamic then
         throw an AttributeError exception — a variable definition cannot have the dynamic attribute
      end if;
      if a.prototype then
         throw an AttributeError exception — a variable definition cannot have the prototype attribute
      end if;
      category: PROPERTYCATEGORY ← a.category;
      if env[0] ∈ CLASS then if category = none then category ← final end if
      else
         if category ≠ none then
            throw an AttributeError exception — non-class variables cannot have a static, virtual, or final
                  attribute
         end if
      end if;
      case category of
         {none, static} do
            initializer: INITIALIZEROPT ← Initializer[VariableInitialisation^β];
            if noInitializer and initializer ≠ none then
               throw a SyntaxError exception — a for-in statement’s variable definition must not have an initialiser
            end if;
            proc variableSetup(): CLASSOPT
               type: CLASSOPT ← SetupAndEval[TypedIdentifier^β](env);
               Evaluate Setup[VariableInitialisation^β]() and ignore its result;
               return type
            end proc;
            v: VARIABLE ← new VARIABLE⟨⟨value: none, immutable: immutable, setup: variableSetup,
                  initializer: initializer, initializerEnv: env⟩⟩;
            multiname: MULTINAME ← defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit,
                  readWrite, v);
            Multiname[VariableBinding^β] ← multiname;
            CompileVar[VariableBinding^β] ← v;
         {virtual, final} do
            note not noInitializer;
            c: CLASS ← env[0];
            v: INSTANCEVARIABLE ← new INSTANCEVARIABLE⟨⟨final: category = final, immutable: immutable⟩⟩;
            vOverridden: INSTANCEVARIABLEOPT ← defineInstanceProperty(c, cxt, name, a.namespaces,
                  a.overrideMod, a.explicit, v);
            enumerable: BOOLEAN ← a.enumerable;
            if vOverridden ≠ none and vOverridden.enumerable then enumerable ← true
            end if;
            v.enumerable ← enumerable;
            OverriddenVar[VariableBinding^β] ← vOverridden;
            CompileVar[VariableBinding^β] ← v
        end case
    end if
end proc;
```

Validate[VariableInitialisation^β] (cxt: Context, env: Environment) propagates the call to Validate to nonterminals in the expansion of VariableInitialisation^β.

Validate[VariableInitializer^β] (cxt: Context, env: Environment) propagates the call to Validate to nonterminals in the expansion of VariableInitializer^β.

```
Name[TypedIdentifier^β]: String;
    Name[TypedIdentifier^β ⇒ Identifier] = Name[Identifier];
    Name[TypedIdentifier^β ⇒ Identifier : TypeExpression^β] = Name[Identifier];

Plain[TypedIdentifier^β]: Boolean;
    Plain[TypedIdentifier^β ⇒ Identifier] = true;
    Plain[TypedIdentifier^β ⇒ Identifier : TypeExpression^β] = false;

proc Validate[TypedIdentifier^β] (cxt: Context, env: Environment)
    [TypedIdentifier^β ⇒ Identifier] do nothing;
    [TypedIdentifier^β ⇒ Identifier : TypeExpression^β] do
        Evaluate Validate[TypeExpression^β](cxt, env) and ignore its result
end proc;
```

Setup

```
proc Setup[VariableDefinition^β ⇒ VariableDefinitionKind VariableBindingList^β] ()
    Evaluate Setup[VariableBindingList^β]() and ignore its result
end proc;
```

Setup[VariableBindingList^β] () propagates the call to Setup to nonterminals in the expansion of VariableBindingList^β.

```
proc Setup[VariableBinding□ □ TypedIdentifier□ VariableInitialisation□] ()
  env: ENVIRONMENT □ CompileEnv[VariableBinding□];
  v: VARIABLE □ DYNAMICVAR □ INSTANCEVARIABLE □ CompileVar[VariableBinding□];
  case v of
    VARIABLE do
      Evaluate setupVariable(v) and ignore its result;
      if not v.immutable then
        defaultValue: OBJECTOPT □ v.type.defaultValue;
        if defaultValue = none then
          throw an UninitializedError exception — Cannot declare a mutable variable of type Never
        end if;
        v.value □ defaultValue
      end if;
    DYNAMICVAR do Evaluate Setup[VariableInitialisation□]() and ignore its result;
    INSTANCEVARIABLE do
      t: CLASSOPT □ SetupAndEval[TypedIdentifier□](env);
      if t = none then
        overriddenVar: INSTANCEVARIABLEOPT □ OverriddenVar[VariableBinding□];
        if overriddenVar ≠ none then t □ overriddenVar.type
        else t □ Object
        end if
      end if;
      v.type □ t;
      Evaluate Setup[VariableInitialisation□]() and ignore its result;
      initializer: INITIALIZEROPT □ Initializer[VariableInitialisation□];
      defaultValue: OBJECTOPT □ none;
      if initializer ≠ none then defaultValue □ initializer(env, compile)
      elsif not v.immutable then
        defaultValue □ t.defaultValue;
        if defaultValue = none then
          throw an UninitializedError exception — Cannot declare a mutable instance variable of type Never
        end if
      end if;
      v.defaultValue □ defaultValue
  end case
end proc;
```

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

Setup[*VariableInitializer*□] () propagates the call to Setup to nonterminals in the expansion of *VariableInitializer*□.

Evaluation

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

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

```
proc Eval[VariableBinding^β ⇒ TypedIdentifier^β VariableInitialisation^β] (env: ENVIRONMENT)
   case CompileVar[VariableBinding^β] of
      VARIABLE do
         innerFrame: NONWITHFRAME ← env[0];
         properties: SINGLETONPROPERTY{} ← {b.content | ∀b ∈ innerFrame.localBindings such that
               b.qname ∈ Multiname[VariableBinding^β]};
         note The properties set consists of exactly one VARIABLE element because innerFrame was constructed with
               that VARIABLE inside Validate.
         v: VARIABLE ← the one element of properties;
         initializer: INITIALIZER ∪ {none, busy} ← v.initializer;
         case initializer of
            {none} do nothing;
            {busy} do throw a ReferenceError exception;
            INITIALIZER do
               v.initializer ← busy;
               value: OBJECT ← initializer(v.initializerEnv, run);
               Evaluate writeVariable(v, value, true) and ignore its result
         end case;
      DYNAMICVAR do
         initializer: INITIALIZEROPT ← Initializer[VariableInitialisation^β];
         if initializer ≠ none then
            value: OBJECT ← initializer(env, run);
            Evaluate lexicalWrite(env, Multiname[VariableBinding^β], value, false, run) and ignore its result
         end if;
      INSTANCEVARIABLE do nothing
   end case
end proc;

proc WriteBinding[VariableBinding^β ⇒ TypedIdentifier^β VariableInitialisation^β]
      (env: ENVIRONMENT, newValue: OBJECT)
   case CompileVar[VariableBinding^β] of
      VARIABLE do
         innerFrame: NONWITHFRAME ← env[0];
         properties: SINGLETONPROPERTY{} ← {b.content | ∀b ∈ innerFrame.localBindings such that
               b.qname ∈ Multiname[VariableBinding^β]};
         note The properties set consists of exactly one VARIABLE element because innerFrame was constructed with
               that VARIABLE inside Validate.
         v: VARIABLE ← the one element of properties;
         Evaluate writeVariable(v, newValue, false) and ignore its result;
      DYNAMICVAR do
         Evaluate lexicalWrite(env, Multiname[VariableBinding^β], newValue, false, run) and ignore its result
   end case
end proc;

Initializer[VariableInitialisation^β]: INITIALIZEROPT;
   Initializer[VariableInitialisation^β ⇒ «empty»] = none;
   Initializer[VariableInitialisation^β ⇒ = VariableInitializer^β] = Eval[VariableInitializer^β];

proc Eval[VariableInitializer^β] (env: ENVIRONMENT, phase: PHASE): OBJECT
   [VariableInitializer^β ⇒ AssignmentExpression^β] do
      return readReference(Eval[AssignmentExpression^β](env, phase), phase);
   [VariableInitializer^β ⇒ AttributeCombination] do
      return Eval[AttributeCombination](env, phase)
end proc;
```

```
proc SetupAndEval[TypedIdentifier^β] (env: ENVIRONMENT): CLASSOPT
  [TypedIdentifier^β → Identifier] do return none;
  [TypedIdentifier^β → Identifier : TypeExpression^β] do
    return SetupAndEval[TypeExpression^β](env)
end proc;
```

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

```
SimpleVariableDefinition → var UntypedVariableBindingList

UntypedVariableBindingList →
    UntypedVariableBinding
  | UntypedVariableBindingList , UntypedVariableBinding

UntypedVariableBinding → Identifier VariableInitialisation^allowIn
```

Validation

```
proc Validate[SimpleVariableDefinition → var UntypedVariableBindingList] (cxt: CONTEXT, env: ENVIRONMENT)
  if cxt.strict or getRegionalFrame(env) ∉ PACKAGE ∪ PARAMETERFRAME then
    throw a SyntaxError exception — a variable may not be defined in a substatement except inside a non-strict
        function or non-strict top-level code; to fix this error, place the definition inside a block
  end if;
  Evaluate Validate[UntypedVariableBindingList](cxt, env) and ignore its result
end proc;
```

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

```
proc Validate[UntypedVariableBinding → Identifier VariableInitialisation^allowIn] (cxt: CONTEXT, env: ENVIRONMENT)
  Evaluate Validate[VariableInitialisation^allowIn](cxt, env) and ignore its result;
  Evaluate defineHoistedVar(env, Name[Identifier], undefined) and ignore its result
end proc;
```

Setup

Setup[*SimpleVariableDefinition*] () propagates the call to Setup to nonterminals in the expansion of *SimpleVariableDefinition*.

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

```
proc Setup[UntypedVariableBinding → Identifier VariableInitialisation^allowIn] ()
  Evaluate Setup[VariableInitialisation^allowIn]() and ignore its result
end proc;
```

Evaluation

```
proc Eval[SimpleVariableDefinition □ var UntypedVariableBindingList] (env: ENVIRONMENT, d: OBJECT): OBJECT
   Evaluate Eval[UntypedVariableBindingList](env) and ignore its result;
   return d
end proc;
```

Eval[*UntypedVariableBindingList*] (*env*: ENVIRONMENT) propagates the call to Eval to nonterminals in the expansion of *UntypedVariableBindingList*.

```
proc Eval[UntypedVariableBinding □ Identifier VariableInitialisation^allowIn] (env: ENVIRONMENT)
   initializer: INITIALIZEROPT □ Initializer[VariableInitialisation^allowIn];
   if initializer ≠ none then
      value: OBJECT □ initializer(env, run);
      qname: QUALIFIEDNAME □ public::(Name[Identifier]);
      Evaluate lexicalWrite(env, {qname}, value, false, run) and ignore its result
   end if
end proc;
```

14.3 Function Definition

Syntax

FunctionDefinition □ **function** *FunctionName FunctionCommon*

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

FunctionCommon □ **(** *Parameters* **)** *Result Block*

Validation

OverriddenProperty[*FunctionDefinition*]: INSTANCEPROPERTYOPT;

```
proc ValidateStatic[FunctionDefinition → function FunctionName FunctionCommon] (cxt: CONTEXT,
      env: ENVIRONMENT, preinst: BOOLEAN, a: COMPOUNDATTRIBUTE, unchecked: BOOLEAN, hoisted: BOOLEAN)
   name: STRING ← Name[FunctionName];
   handling: HANDLING ← Handling[FunctionName];
   case handling of
      {normal} do
         kind: STATICFUNCTIONKIND;
         if unchecked then kind ← uncheckedFunction
         elsif a.prototype then kind ← prototypeFunction
         else kind ← plainFunction
         end if;
         f: SIMPLEINSTANCE ∪ UNINSTANTIATEDFUNCTION ←
               ValidateStaticFunction[FunctionCommon](cxt, env, kind);
         if preinst then f ← instantiateFunction(f, env) end if;
         if hoisted then Evaluate defineHoistedVar(env, name, f) and ignore its result
         else
            v: VARIABLE ← new VARIABLE〈〈type: Function, value: f, immutable: true, setup: none, initializer: none〉〉;
            Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v) and
                  ignore its result
         end if;
      {get, set} do
         if a.prototype then
            throw an AttributeError exception — a getter or setter cannot have the prototype attribute
         end if;
         note  not (unchecked or hoisted);
         Evaluate Validate[FunctionCommon](cxt, env, plainFunction, handling) and ignore its result;
         boundEnv: ENVIRONMENTOPT ← none;
         if preinst then boundEnv ← env end if;
         case handling of
            {get} do
               getter: GETTER ← new GETTER〈〈call: EvalStaticGet[FunctionCommon], env: boundEnv〉〉;
               Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, read, getter)
                     and ignore its result;
            {set} do
               setter: SETTER ← new SETTER〈〈call: EvalStaticSet[FunctionCommon], env: boundEnv〉〉;
               Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, write, setter)
                     and ignore its result
         end case
   end case;
   OverriddenProperty[FunctionDefinition] ← none
end proc;
```

```
proc ValidateInstance[FunctionDefinition ⇒ function FunctionName FunctionCommon]
      (cxt: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDATTRIBUTE, final: BOOLEAN)
   if a.prototype then
      throw an AttributeError exception — an instance method cannot have the prototype attribute
   end if;
   handling: HANDLING ← Handling[FunctionName];
   Evaluate Validate[FunctionCommon](cxt, env, instanceFunction, handling) and ignore its result;
   signature: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   m: INSTANCEPROPERTY;
   case handling of
      {normal} do
         m ← new INSTANCEMETHOD⟨⟨final: final, signature: signature, length: signatureLength(signature),
                call: EvalInstanceCall[FunctionCommon]⟩⟩;
      {get} do
         m ← new INSTANCEGETTER⟨⟨final: final, signature: signature, call: EvalInstanceGet[FunctionCommon]⟩⟩;
      {set} do
         m ← new INSTANCESETTER⟨⟨final: final, signature: signature, call: EvalInstanceSet[FunctionCommon]⟩⟩
   end case;
   mOverridden: INSTANCEPROPERTYOPT ← defineInstanceProperty(c, cxt, Name[FunctionName], a.namespaces,
         a.overrideMod, a.explicit, m);
   enumerable: BOOLEAN ← a.enumerable;
   if mOverridden ≠ none and mOverridden.enumerable then enumerable ← true end if;
   m.enumerable ← enumerable;
   OverriddenProperty[FunctionDefinition] ← mOverridden
end proc;

proc ValidateConstructor[FunctionDefinition ⇒ function FunctionName FunctionCommon]
      (cxt: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDATTRIBUTE)
   if a.prototype then
      throw an AttributeError exception — a class constructor cannot have the prototype attribute
   end if;
   if Handling[FunctionName] ∈ {get, set} then
      throw a SyntaxError exception — a class constructor cannot be a getter or a setter
   end if;
   Evaluate Validate[FunctionCommon](cxt, env, constructorFunction, normal) and ignore its result;
   if c.init ≠ none then
      throw a DefinitionError exception — duplicate constructor definition
   end if;
   c.init ← EvalInstanceInit[FunctionCommon];
   OverriddenProperty[FunctionDefinition] ← none
end proc;
```

```
proc Validate[FunctionDefinition ⇒ function FunctionName FunctionCommon]
      (cxt: Context, env: Environment, preinst: Boolean, attr: AttributeOptNotFalse)
   a: CompoundAttribute ← toCompoundAttribute(attr);
   if a.dynamic then
      throw an AttributeError exception — a function cannot have the dynamic attribute
   end if;
   frame: Frame ← env[0];
   if frame ∈ Class then
      note preinst;
      case a.category of
         {static} do
            Evaluate ValidateStatic[FunctionDefinition](cxt, env, preinst, a, false, false) and ignore its result;
         {none} do
            if Name[FunctionName] = frame.name then
               Evaluate ValidateConstructor[FunctionDefinition](cxt, env, frame, a) and ignore its result
            else
               Evaluate ValidateInstance[FunctionDefinition](cxt, env, frame, a, false) and ignore its result
            end if;
         {virtual} do
            Evaluate ValidateInstance[FunctionDefinition](cxt, env, frame, a, false) and ignore its result;
         {final} do
            Evaluate ValidateInstance[FunctionDefinition](cxt, env, frame, a, true) and ignore its result
      end case
   else
      if a.category ≠ none then
         throw an AttributeError exception — non-class functions cannot have a static, virtual, or final
               attribute
      end if;
      unchecked: Boolean ← not cxt.strict and Handling[FunctionName] = normal and Plain[FunctionCommon];
      hoisted: Boolean ← unchecked and attr = none and
            (frame ∈ Package or (frame ∈ LocalFrame and env[1] ∈ ParameterFrame));
      Evaluate ValidateStatic[FunctionDefinition](cxt, env, preinst, a, unchecked, hoisted) and ignore its result
   end if
end proc;
```

```
Handling[FunctionName]: Handling;
   Handling[FunctionName ⇒ Identifier] = normal;
   Handling[FunctionName ⇒ get [no line break] Identifier] = get;
   Handling[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, kind: FunctionKind, handling: Handling)
    localCxt: Context ← new Context⟨strict: cxt.strict, openNamespaces: cxt.openNamespaces⟩;
    superconstructorCalled: Boolean ← kind ≠ constructorFunction;
    compileFrame: ParameterFrame ← new ParameterFrame⟨localBindings: {}, kind: kind, handling: handling,
            callsSuperconstructor: false, superconstructorCalled: superconstructorCalled, this: none, parameters: [],
            rest: none⟩;
    compileEnv: Environment ← [compileFrame] ⊕ env;
    CompileFrame[FunctionCommon] ← compileFrame;
    CompileEnv[FunctionCommon] ← compileEnv;
    if kind = uncheckedFunction then
        Evaluate defineHoistedVar(compileEnv, "arguments", undefined) and ignore its result
    end if;
    Evaluate Validate[Parameters](localCxt, compileEnv, compileFrame) and ignore its result;
    Evaluate Validate[Result](localCxt, compileEnv) and ignore its result;
    Evaluate Validate[Block](localCxt, compileEnv, JumpTargets⟨breakTargets: {}, continueTargets: {}⟩, false) and
            ignore its result
end proc;

proc ValidateStaticFunction[FunctionCommon ⇒ ( Parameters ) Result Block]
        (cxt: Context, env: Environment, kind: StaticFunctionKind): UninstantiatedFunction
    Evaluate Validate[FunctionCommon](cxt, env, kind, normal) and ignore its result;
    length: Integer ← ParameterCount[Parameters];
    case kind of
        {plainFunction} do
            return new UninstantiatedFunction⟨type: Function, length: length,
                    call: EvalStaticCall[FunctionCommon], construct: none, instantiations: {}⟩
        {uncheckedFunction, prototypeFunction} do
            return new UninstantiatedFunction⟨type: PrototypeFunction, length: length,
                    call: EvalStaticCall[FunctionCommon], construct: EvalPrototypeConstruct[FunctionCommon],
                    instantiations: {}⟩
    end case
end proc;
```

Setup

```
proc Setup[FunctionDefinition ⇒ function FunctionName FunctionCommon] ()
  overriddenProperty: INSTANCEPROPERTYOPT ← OverriddenProperty[FunctionDefinition];
  case overriddenProperty of
    {none} do Evaluate Setup[FunctionCommon]() and ignore its result;
    INSTANCEMETHOD ∪ INSTANCEGETTER ∪ INSTANCESETTER do
      Evaluate SetupOverride[FunctionCommon](overriddenProperty.signature) and ignore its result;
    INSTANCEVARIABLE do
      overriddenSignature: PARAMETERFRAME;
      case Handling[FunctionName] of
        {normal} do
          This cannot happen because ValidateInstance already ensured that a function cannot override an
          instance variable.
        {get} do
          overriddenSignature ← new PARAMETERFRAME⟨localBindings: {}, kind: instanceFunction,
                handling: get, callsSuperconstructor: false, superconstructorCalled: false, this: none,
                parameters: [], rest: none, returnType: overriddenProperty.type⟩
        {set} do
          v: VARIABLE ← new VARIABLE⟨type: overriddenProperty.type, value: none, immutable: false,
                setup: none, initializer: none⟩
          parameters: PARAMETER[] ← [PARAMETER⟨var: v, default: none⟩];
          overriddenSignature ← new PARAMETERFRAME⟨localBindings: {}, kind: instanceFunction,
                handling: set, callsSuperconstructor: false, superconstructorCalled: false, this: none,
                parameters: parameters, rest: none, returnType: Void⟩
      end case;
      Evaluate SetupOverride[FunctionCommon](overriddenSignature) and ignore its result
  end case
end proc;

proc Setup[FunctionCommon ⇒ ( Parameters ) Result Block] ()
  compileEnv: ENVIRONMENT ← CompileEnv[FunctionCommon];
  compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
  Evaluate Setup[Parameters](compileEnv, compileFrame) and ignore its result;
  Evaluate checkAccessorParameters(compileFrame) and ignore its result;
  Evaluate Setup[Result](compileEnv, compileFrame) and ignore its result;
  Evaluate Setup[Block]() and ignore its result
end proc;

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

Evaluation

```
proc EvalStaticCall[FunctionCommon □ ( Parameters ) Result Block]
      (this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
   note  The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call user-defined functions
   end if;
   runtimeEnv: ENVIRONMENT □ f.env;
   runtimeThis: OBJECTOPT □ none;
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   if compileFrame.kind □ {uncheckedFunction, prototypeFunction} then
      if this □ PRIMITIVEOBJECT then runtimeThis □ getPackageFrame(runtimeEnv)
      else runtimeThis □ this
      end if
   end if;
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
   Evaluate assignArguments(runtimeFrame, f, args, phase) and ignore its result;
   result: OBJECT;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined) and ignore its result;
      result □ undefined
   catch x: SEMANTICEXCEPTION do
      if x □ RETURN then result □ x.value else throw x end if
   end try;
   return coerce(result, runtimeFrame.returnType)
end proc;

proc EvalStaticGet[FunctionCommon □ ( Parameters ) Result Block]
      (runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
   note  The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call user-defined getters
   end if;
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, runtimeEnv, none);
   Evaluate assignArguments(runtimeFrame, none, [], phase) and ignore its result;
   result: OBJECT;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined) and ignore its result;
      throw a SyntaxError exception — a getter must return a value and may not return by falling off the end of its code
   catch x: SEMANTICEXCEPTION do
      if x □ RETURN then result □ x.value else throw x end if
   end try;
   return coerce(result, runtimeFrame.returnType)
end proc;
```

```
proc EvalStaticSet[FunctionCommon ⇒ ( Parameters ) Result Block]
      (newValue: OBJECT, runtimeEnv: ENVIRONMENT, phase: PHASE)
   note The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call setters
   end if;
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, runtimeEnv, none);
   Evaluate assignArguments(runtimeFrame, none, [newValue], phase) and ignore its result;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined) and ignore its result
   catch x: SEMANTICEXCEPTION do if x ∉ RETURN then throw x end if
   end try
end proc;

proc EvalInstanceCall[FunctionCommon ⇒ ( Parameters ) Result Block]
      (this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
   note The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call user-defined functions
   end if;
   note Class frames are always preinstantiated, so the run environment is the same as compile environment.
   env: ENVIRONMENT ← CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, env, this);
   Evaluate assignArguments(runtimeFrame, none, args, phase) and ignore its result;
   result: OBJECT;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ env, undefined) and ignore its result;
      result ← undefined
   catch x: SEMANTICEXCEPTION do
      if x ∈ RETURN then result ← x.value else throw x end if
   end try;
   return coerce(result, runtimeFrame.returnType)
end proc;

proc EvalInstanceGet[FunctionCommon ⇒ ( Parameters ) Result Block] (this: OBJECT, phase: PHASE): OBJECT
   note The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call user-defined getters
   end if;
   note Class frames are always preinstantiated, so the run environment is the same as compile environment.
   env: ENVIRONMENT ← CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME ← CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME ← instantiateParameterFrame(compileFrame, env, this);
   Evaluate assignArguments(runtimeFrame, none, [], phase) and ignore its result;
   result: OBJECT;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ env, undefined) and ignore its result;
      throw a SyntaxError exception — a getter must return a value and may not return by falling off the end of its code
   catch x: SEMANTICEXCEPTION do
      if x ∈ RETURN then result ← x.value else throw x end if
   end try;
   return coerce(result, runtimeFrame.returnType)
end proc;
```

```
proc EvalInstanceSet[FunctionCommon □ ( Parameters ) Result Block]
      (this: OBJECT, newValue: OBJECT, phase: PHASE)
   note  The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call setters
   end if;
   note  Class frames are always preinstantiated, so the run environment is the same as compile environment.
   env: ENVIRONMENT □ CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, env, this);
   Evaluate assignArguments(runtimeFrame, none, [newValue], phase) and ignore its result;
   try Evaluate Eval[Block]([runtimeFrame] ⊕ env, undefined) and ignore its result
   catch x: SEMANTICEXCEPTION do if x □ RETURN then throw x end if
   end try
end proc;

proc EvalInstanceInit[FunctionCommon □ ( Parameters ) Result Block]
      (this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
   note  Class frames are always preinstantiated, so the run environment is the same as compile environment.
   env: ENVIRONMENT □ CompileEnv[FunctionCommon];
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, env, this);
   Evaluate assignArguments(runtimeFrame, none, args, phase) and ignore its result;
   if not runtimeFrame.callsSuperconstructor then
      c: CLASS □ getEnclosingClass(env);
      Evaluate callInit(this, c.super, [], run) and ignore its result;
      runtimeFrame.superconstructorCalled □ true
   end if;
   try Evaluate Eval[Block]([runtimeFrame] ⊕ env, undefined) and ignore its result
   catch x: SEMANTICEXCEPTION do if x □ RETURN then throw x end if
   end try;
   if not runtimeFrame.superconstructorCalled then
      throw an UninitializedError exception — the superconstructor must be called before returning normally from a
            constructor
   end if
end proc;
```

```
proc EvalPrototypeConstruct[FunctionCommon □ ( Parameters ) Result Block]
      (f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
   note  The check that phase ≠ compile also ensures that Setup has been called.
   if phase = compile then
      throw a ConstantError exception — a constant expression cannot call user-defined prototype constructors
   end if;
   runtimeEnv: ENVIRONMENT □ f.env;
   archetype: OBJECT □ dotRead(f, {public::“prototype”}, phase);
   if archetype □ {null, undefined} then archetype □ ObjectPrototype
   elsif objectType(archetype) ≠ Object then
      throw a TypeError exception — bad prototype value
   end if;
   o: OBJECT □ createSimpleInstance(Object, archetype, none, none, none);
   compileFrame: PARAMETERFRAME □ CompileFrame[FunctionCommon];
   runtimeFrame: PARAMETERFRAME □ instantiateParameterFrame(compileFrame, runtimeEnv, o);
   Evaluate assignArguments(runtimeFrame, f, args, phase) and ignore its result;
   result: OBJECT;
   try
      Evaluate Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined) and ignore its result;
      result □ undefined
   catch x: SEMANTICEXCEPTION do
      if x □ RETURN then result □ x.value else throw x end if
   end try;
   coercedResult: OBJECT □ coerce(result, runtimeFrame.returnType);
   if coercedResult □ PRIMITIVEOBJECT then return o else return coercedResult end if
end proc;

proc checkAccessorParameters(frame: PARAMETERFRAME)
   parameters: PARAMETER[] □ frame.parameters;
   rest: VARIABLEOPT □ frame.rest;
   case frame.handling of
      {normal} do nothing;
      {get} do
         if parameters ≠ [] or rest ≠ none then
            throw a SyntaxError exception — a getter cannot take any parameters
         end if;
      {set} do
         if |parameters| ≠ 1 or rest ≠ none then
            throw a SyntaxError exception — a setter must take exactly one parameter
         end if;
         if parameters[0].default ≠ none then
            throw a SyntaxError exception — a setter’s parameter cannot be optional
         end if
   end case
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.kind = uncheckedFunction then
      argumentsObject ← construct(Array, [], phase);
      Evaluate createDynamicProperty(argumentsObject, public::“callee”, false, false, f) and ignore its result;
      Evaluate writeArrayPrivateLength(argumentsObject, |args|, phase) and ignore its result
   end if;
   restObject: OBJECTOPT ← none;
   rest: VARIABLE ∪ {none} ← runtimeFrame.rest;
   if rest ≠ none then restObject ← construct(Array, [], 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];
         default: OBJECTOPT ← parameter.default;
         argOrDefault: OBJECT ← arg;
         if argOrDefault = undefined and default ≠ none then argOrDefault ← default
         end if;
         v: DYNAMICVAR ∪ VARIABLE ← parameter.var;
         Evaluate writeSingletonProperty(v, argOrDefault, phase) and ignore its result;
         if argumentsObject ≠ none then
            note Create an alias of v as the ith entry of the arguments object.
            note v ∈ DYNAMICVAR;
            qname: QUALIFIEDNAME ← objectToQualifiedName(i_f64, phase);
            argumentsObject.localBindings ← argumentsObject.localBindings ∪ {LOCALBINDING⟨⟨qname: qname,
                  accesses: readWrite, explicit: false, enumerable: false, content: v⟩⟩}
         end if
      elsif restObject ≠ none then
         if j ≥ arrayLimit then throw a RangeError exception end if;
         Evaluate indexWrite(restObject, j, arg, phase) and ignore its result;
         note argumentsObject = none because a function can't have both a rest parameter and an arguments object.
         j ← j + 1
      elsif argumentsObject ≠ none then
         Evaluate indexWrite(argumentsObject, i, arg, phase) and ignore its result
      else
         throw an ArgumentError exception — more arguments than parameters were supplied, and the called function
               does not have a ... parameter and is not unchecked.
      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 an ArgumentError exception — fewer arguments than parameters were supplied, and the called
                    function does not supply default values for the missing parameters and is not unchecked.
            end if
        end if;
        Evaluate writeSingletonProperty(parameter.var, default, phase) and ignore its result;
        i ← i + 1
    end while
end proc;

proc signatureLength(signature: PARAMETERFRAME): INTEGER
    return |signature.parameters|
end proc;
```

Syntax

```
Parameters ⇒
    «empty»
  | NonemptyParameters

NonemptyParameters ⇒
    ParameterInit
  | ParameterInit , NonemptyParameters
  | RestParameter

Parameter ⇒ ParameterAttributes TypedIdentifier^allowIn

ParameterAttributes ⇒
    «empty»
  | const

ParameterInit ⇒
    Parameter
  | Parameter = AssignmentExpression^allowIn

RestParameter ⇒
    ...
  | ... ParameterAttributes Identifier

Result ⇒
    «empty»
  | : TypeExpression^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 nonterminals 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 nonterminals 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)
   Evaluate Validate[TypedIdentifier^allowIn](cxt, env) and ignore its result;
   immutable: BOOLEAN ← HasConst[ParameterAttributes];
   name: STRING ← Name[TypedIdentifier^allowIn];
   v: DYNAMICVAR ∪ VARIABLE;
   if compileFrame ∈ PARAMETERFRAME and compileFrame.kind = uncheckedFunction then
      note not immutable;
      v ← defineHoistedVar(env, name, undefined)
   else
      v ← new VARIABLE⟨⟨value: none, immutable: immutable, setup: none, initializer: none⟩⟩;
      Evaluate defineSingletonProperty(env, name, {public}, none, false, readWrite, v) and ignore its result
   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
      Evaluate Validate[Parameter](cxt, env, compileFrame) and ignore its result;
   [ParameterInit ⇒ Parameter = AssignmentExpression^allowIn] do
      Evaluate Validate[Parameter](cxt, env, compileFrame) and ignore its result;
      Evaluate Validate[AssignmentExpression^allowIn](cxt, env) and ignore its result
end proc;
```

```
proc Validate[RestParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
   [RestParameter → ...] do
      note compileFrame.kind ≠ uncheckedFunction;
      v: VARIABLE ← new VARIABLE⟨type: Array, value: none, immutable: true, setup: none, initializer: none⟩;
      compileFrame.rest ← v;
   [RestParameter → ... ParameterAttributes Identifier] do
      note compileFrame.kind ≠ uncheckedFunction;
      v: VARIABLE ← new VARIABLE⟨type: Array, value: none, immutable: HasConst[ParameterAttributes],
            setup: none, initializer: none⟩;
      compileFrame.rest ← v;
      name: STRING ← Name[Identifier];
      Evaluate defineSingletonProperty(env, name, {public}, none, false, readWrite, v) and ignore its result
end proc;
```

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

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

Setup

Setup[*Parameters*] (*compileEnv*: ENVIRONMENT, *compileFrame*: PARAMETERFRAME) propagates the call to Setup to nonterminals 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 a DefinitionError exception — mismatch with the overridden method's signature
      end if;
   [Parameters → NonemptyParameters] do
      Evaluate SetupOverride[NonemptyParameters](compileEnv, compileFrame, overriddenSignature,
            overriddenSignature.parameters) and ignore its result
end proc;
```

```
proc Setup[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
   [NonemptyParameters → ParameterInit] do
      Evaluate Setup[ParameterInit](compileEnv, compileFrame) and ignore its result;
   [NonemptyParameters_0 → ParameterInit , NonemptyParameters_1] do
      Evaluate Setup[ParameterInit](compileEnv, compileFrame) and ignore its result;
      Evaluate Setup[NonemptyParameters_1](compileEnv, compileFrame) and ignore its result;
   [NonemptyParameters → RestParameter] do nothing
end proc;
```

```
proc SetupOverride[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
    overriddenSignature: PARAMETERFRAME, overriddenParameters: PARAMETER[])
  [NonemptyParameters ⇒ ParameterInit] do
    if overriddenParameters = [] then
      throw a DefinitionError exception — mismatch with the overridden method’s signature
    end if;
    Evaluate SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]) and ignore its
        result;
    if |overriddenParameters| ≠ 1 or overriddenSignature.rest ≠ none then
      throw a DefinitionError exception — mismatch with the overridden method’s signature
    end if;
  [NonemptyParameters_0 ⇒ ParameterInit , NonemptyParameters_1] do
    if overriddenParameters = [] then
      throw a DefinitionError exception — mismatch with the overridden method’s signature
    end if;
    Evaluate SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]) and ignore its
        result;
    Evaluate SetupOverride[NonemptyParameters_1](compileEnv, compileFrame, overriddenSignature,
        overriddenParameters[1 ...]) and ignore its result;
  [NonemptyParameters ⇒ RestParameter] do
    if overriddenParameters ≠ [] then
      throw a DefinitionError exception — mismatch with the overridden method’s signature
    end if;
    overriddenRest: VARIABLE ∪ {none} ← overriddenSignature.rest;
    if overriddenRest = none or overriddenRest.type ≠ Array then
      throw a DefinitionError exception — mismatch with the overridden method’s signature
    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
    throw a SyntaxError exception — a required parameter cannot follow an optional one
  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 ← Object 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
        throw a SyntaxError exception — a required parameter cannot follow an optional one
    end if;
    v: DYNAMICVAR ∪ VARIABLE ← CompileVar[Parameter];
    note v ∈ DYNAMICVAR;
    type: CLASSOPT ← SetupAndEval[TypedIdentifier$^{allowIn}$](compileEnv);
    if type = none then type ← Object end if;
    if type ≠ overriddenParameter.var.type then
        throw a DefinitionError exception — mismatch with the overridden method’s signature
    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
        Evaluate Setup[Parameter](compileEnv, compileFrame, none) and ignore its result;
    [ParameterInit ⇒ Parameter = AssignmentExpression$^{allowIn}$] do
        Evaluate Setup[AssignmentExpression$^{allowIn}$]() and ignore its result;
        default: OBJECT ← readReference(Eval[AssignmentExpression$^{allowIn}$](compileEnv, compile), compile);
        Evaluate Setup[Parameter](compileEnv, compileFrame, default) and ignore its result
end proc;

proc SetupOverride[ParameterInit]
        (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenParameter: PARAMETER)
    [ParameterInit ⇒ Parameter] do
        Evaluate SetupOverride[Parameter](compileEnv, compileFrame, none, overriddenParameter) and ignore its
                result;
    [ParameterInit ⇒ Parameter = AssignmentExpression$^{allowIn}$] do
        Evaluate Setup[AssignmentExpression$^{allowIn}$]() and ignore its result;
        default: OBJECT ← readReference(Eval[AssignmentExpression$^{allowIn}$](compileEnv, compile), compile);
        Evaluate SetupOverride[Parameter](compileEnv, compileFrame, default, overriddenParameter) and ignore its
                result
end proc;

proc Setup[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [Result ⇒ «empty»] do
        defaultReturnType: CLASS ← Object;
        if cannotReturnValue(compileFrame) then defaultReturnType ← Void end if;
        compileFrame.returnType ← defaultReturnType;
    [Result ⇒ : TypeExpression$^{allowIn}$] do
        if cannotReturnValue(compileFrame) then
            throw a SyntaxError exception — a setter or constructor cannot define a return type
        end if;
        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 a DefinitionError exception — mismatch with the overridden method's signature
        end if;
        compileFrame.returnType ← t
end proc;
```

14.4 Class Definition

Syntax

ClassDefinition ⇒ **`class`** *Identifier Inheritance Block*

Inheritance ⇒
  «empty»
| **`extends`** *TypeExpression*$^{\text{allowIn}}$

Validation

Class[*ClassDefinition*]: CLASS;

```
proc Validate[ClassDefinition → class Identifier Inheritance Block]
      (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
   if not preinst then
      throw a SyntaxError exception — a class may be defined only in a preinstantiated scope
   end if;
   super: CLASS ← Validate[Inheritance](cxt, env);
   if not super.complete then
      throw a ConstantError exception — cannot override a class before its definition has been compiled
   end if;
   if super.final then throw a DefinitionError exception — can't override a final class
   end if;
   a: COMPOUNDATTRIBUTE ← toCompoundAttribute(attr);
   if a.prototype then
      throw an AttributeError exception — a class definition cannot have the prototype attribute
   end if;
   final: BOOLEAN;
   case a.category of
      {none} do final ← false;
      {static} do
         if env[0] ∉ CLASS then
            throw an AttributeError exception — non-class property definitions cannot have a static attribute
         end if;
         final ← false;
      {final} do final ← true;
      {virtual} do
         throw an AttributeError exception — a class definition cannot have the virtual attribute
   end case;
   privateNamespace: NAMESPACE ← new NAMESPACE⟨⟨name: “private”⟩⟩;
   dynamic: BOOLEAN ← a.dynamic or (super.dynamic and super ≠ Object);
   c: CLASS ← new CLASS⟨⟨localBindings: {}, instanceProperties: {}, super: super, prototype: super.prototype,
         complete: false, name: Name[Identifier], typeofString: “object”, privateNamespace: privateNamespace,
         dynamic: dynamic, final: final, defaultValue: null, defaultHint: hintNumber,
         hasProperty: super.hasProperty, bracketRead: super.bracketRead, bracketWrite: super.bracketWrite,
         bracketDelete: super.bracketDelete, read: super.read, write: super.write, delete: super.delete,
         enumerate: super.enumerate, call: ordinaryCall, construct: ordinaryConstruct, init: none, is: ordinaryIs,
         coerce: ordinaryCoerce⟩⟩
   Class[ClassDefinition] ← c;
   v: VARIABLE ← new VARIABLE⟨⟨type: Class, value: c, immutable: true, setup: none, initializer: none⟩⟩
   Evaluate defineSingletonProperty(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v)
         and ignore its result;
   innerCxt: CONTEXT ← new CONTEXT⟨⟨strict: cxt.strict,
         openNamespaces: cxt.openNamespaces ∪ {privateNamespace}⟩⟩
   Evaluate ValidateUsingFrame[Block](innerCxt, env, JUMPTARGETS⟨⟨breakTargets: {}, continueTargets: {}⟩⟩,
         preinst, c) and ignore its result;
   if c.init = none then c.init ← super.init end if;
   c.complete ← true
end proc;

proc Validate[Inheritance] (cxt: CONTEXT, env: ENVIRONMENT): CLASS
   [Inheritance → «empty»] do return Object;
   [Inheritance → extends TypeExpression^allowIn] do
      Evaluate Validate[TypeExpression^allowIn](cxt, env) and ignore its result;
      return SetupAndEval[TypeExpression^allowIn](env)
end proc;
```

Setup

```
proc Setup[ClassDefinition → class Identifier Inheritance Block] ()
   Evaluate Setup[Block]() and ignore its result
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;
```

14.5 Namespace Definition

Syntax

NamespaceDefinition → **namespace** *Identifier*

Validation

```
proc Validate[NamespaceDefinition → namespace Identifier]
      (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
   if not preinst then
      throw a SyntaxError exception — a namespace may be defined only in a preinstantiated scope
   end if;
   a: COMPOUNDATTRIBUTE ← toCompoundAttribute(attr);
   if a.dynamic then
      throw an AttributeError exception — a namespace definition cannot have the dynamic attribute
   end if;
   if a.prototype then
      throw an AttributeError exception — a namespace definition cannot have the prototype attribute
   end if;
   case a.category of
      {none} do nothing;
      {static} do
         if env[0] ∉ CLASS then
            throw an AttributeError exception — non-class property definitions cannot have a static attribute
         end if;
      {virtual, final} do
         throw an AttributeError exception — a namespace definition cannot have the virtual or final attribute
   end case;
   name: STRING ← Name[Identifier];
   ns: NAMESPACE ← new NAMESPACE⟨⟨name: name⟩⟩;
   v: VARIABLE ← new VARIABLE⟨⟨type: Namespace, value: ns, immutable: true, setup: none, initializer: none⟩⟩;
   Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v) and ignore
         its result
end proc;
```

~~16~~15 Programs

Syntax

```
Program ⇒
    Directives
  | PackageDefinition Program
```

Processing

```
Process[Program]: OBJECT;
   Process[Program ⇒ Directives]
      begin
         cxt: CONTEXT ← new CONTEXT⟨⟨strict: false, openNamespaces: {public, internal}⟩⟩;
         initialEnvironment: ENVIRONMENT ← [createGlobalObject()];
         Evaluate Validate[Directives](cxt, initialEnvironment,
               JUMPTARGETS⟨⟨breakTargets: {}, continueTargets: {}⟩⟩, true, none) and ignore its result;
         Evaluate Setup[Directives]() and ignore its result;
         return Eval[Directives](initialEnvironment, undefined)
      end;
   Process[Program0 ⇒ PackageDefinition Program1]
      begin
         Evaluate Process[PackageDefinition] and ignore its result;
         return Process[Program1]
      end;
```

15.1 Package Definition

Syntax

```
PackageDefinition ⇒ package PackageNameOpt Block

PackageNameOpt ⇒
    «empty»
  | PackageName

PackageName ⇒
    String
  | PackageIdentifiers

PackageIdentifiers ⇒
    Identifier
  | PackageIdentifiers . Identifier
```

Processing

```
Process[PackageDefinition ⇒ package PackageNameOpt Block]: VOID
  begin
    name: STRING ← Name[PackageNameOpt];
    cxt: CONTEXT ← new CONTEXT⟨⟨strict: false, openNamespaces: {public, internal}⟩⟩;
    globalObject: PACKAGE ← createGlobalObject();
    pkgInternal: NAMESPACE ← new NAMESPACE⟨⟨name: “internal”⟩⟩;
    pkg: PACKAGE ← new PACKAGE⟨⟨localBindings:
          {stdExplicitConstBinding(internal::“internal”, Namespace, internal)}, archetype: ObjectPrototype,
          name: name, initialize: busy, sealed: true, internalNamespace: pkgInternal⟩⟩;
    initialEnvironment: ENVIRONMENT ← [pkg, globalObject];
    Evaluate Validate[Block](cxt, initialEnvironment, JUMPTARGETS⟨⟨breakTargets: {}, continueTargets: {}⟩⟩, true)
          and ignore its result;
    Evaluate Setup[Block]() and ignore its result;
    proc evalPackage()
      pkg.initialize ← busy;
      Evaluate Eval[Block](initialEnvironment, undefined) and ignore its result;
      pkg.initialize ← none
    end proc;
    pkg.initialize ← evalPackage;
    Bind name to package pkg in the system’s list of packages in an implementation-defined manner.
  end;

Name[PackageNameOpt]: STRING;
  Name[PackageNameOpt ⇒ «empty»] = an implementation-supplied name;
  Name[PackageNameOpt ⇒ PackageName] = Name[PackageName];

Name[PackageName]: STRING;
  Name[PackageName ⇒ String] = Value[String] processed in an implementation-defined manner;
  Name[PackageName ⇒ PackageIdentifiers] = Names[PackageIdentifiers] processed in an implementation-defined
        manner;

Names[PackageIdentifiers]: STRING[];
  Names[PackageIdentifiers ⇒ Identifier] = [Name[Identifier]];
  Names[PackageIdentifiers_0 ⇒ PackageIdentifiers_1 . Identifier] = Names[PackageIdentifiers_1] ⊕ [Name[Identifier]];

packageDatabase: PACKAGE{} ← {};
```

- 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.

16 Predefined Identifiers

```
proc createGlobalObject(): PACKAGE
```

```
return new PACKAGE⟦localBindings: {
        stdExplicitConstBinding(internal::“internal”, Namespace, internal),
        stdConstBinding(public::“explicit”, Attribute, global_explicit),
        stdConstBinding(public::“enumerable”, Attribute, global_enumerable),
        stdConstBinding(public::“dynamic”, Attribute, global_dynamic),
        stdConstBinding(public::“static”, Attribute, global_static),
        stdConstBinding(public::“virtual”, Attribute, global_virtual),
        stdConstBinding(public::“final”, Attribute, global_final),
        stdConstBinding(public::“prototype”, Attribute, global_prototype),
        stdConstBinding(public::“unused”, Attribute, global_unused),
        stdFunction(public::“override”, global_override, 1),
        stdConstBinding(public::“NaN”, Number, $\mathbf{NaN}_{f64}$),
        stdConstBinding(public::“Infinity”, Number, $+\infty_{f64}$),
        stdConstBinding(public::“fNaN”, float, $\mathbf{NaN}_{f32}$),
        stdConstBinding(public::“fInfinity”, float, $+\infty_{f32}$),
        stdConstBinding(public::“undefined”, Void, undefined),
        stdFunction(public::“eval”, global_eval, 1),
        stdFunction(public::“parseInt”, global_parseint, 2),
        stdFunction(public::“parseLong”, global_parselong, 2),
        stdFunction(public::“parseFloat”, global_parsefloat, 1),
        stdFunction(public::“isNaN”, global_isnan, 1),
        stdFunction(public::“isFinite”, global_isfinite, 1),
        stdFunction(public::“decodeURI”, global_decodeuri, 1),
        stdFunction(public::“decodeURIComponent”, global_decodeuricomponent, 1),
        stdFunction(public::“encodeURI”, global_encodeuri, 1),
        stdFunction(public::“encodeURIComponent”, global_encodeuricomponent, 1),
        stdConstBinding(public::“Object”, Class, Object),
        stdConstBinding(public::“Never”, Class, Never),
        stdConstBinding(public::“Void”, Class, Void),
        stdConstBinding(public::“Null”, Class, Null),
        stdConstBinding(public::“Boolean”, Class, Boolean),
        stdConstBinding(public::“GeneralNumber”, Class, GeneralNumber),
        stdConstBinding(public::“long”, Class, long),
        stdConstBinding(public::“ulong”, Class, ulong),
        stdConstBinding(public::“float”, Class, float),
        stdConstBinding(public::“Number”, Class, Number),
        stdConstBinding(public::“sbyte”, Class, sbyte),
        stdConstBinding(public::“byte”, Class, byte),
        stdConstBinding(public::“short”, Class, short),
        stdConstBinding(public::“ushort”, Class, ushort),
        stdConstBinding(public::“int”, Class, int),
        stdConstBinding(public::“uint”, Class, uint),
        stdConstBinding(public::“char”, Class, char),
        stdConstBinding(public::“String”, Class, String),
        stdConstBinding(public::“Array”, Class, Array),
        stdConstBinding(public::“Namespace”, Class, Namespace),
        stdConstBinding(public::“Attribute”, Class, Attribute),
        stdConstBinding(public::“Date”, Class, Date),
        stdConstBinding(public::“RegExp”, Class, RegExp),
        stdConstBinding(public::“Class”, Class, Class),
        stdConstBinding(public::“Function”, Class, Function),
        stdConstBinding(public::“PrototypeFunction”, Class, PrototypeFunction),
        stdConstBinding(public::“Package”, Class, Package),
        stdConstBinding(public::“Error”, Class, Error),
            stdConstBinding(public::“ArgumentError”, Class, ArgumentError),
            stdConstBinding(public::“AttributeError”, Class, AttributeError),
            stdConstBinding(public::“ConstantError”, Class, ConstantError),
            stdConstBinding(public::“DefinitionError”, Class, DefinitionError),
            stdConstBinding(public::“EvalError”, Class, EvalError),
            stdConstBinding(public::“RangeError”, Class, RangeError),
            stdConstBinding(public::“ReferenceError”, Class, ReferenceError),
            stdConstBinding(public::“SyntaxError”, Class, SyntaxError),
            stdConstBinding(public::“TypeError”, Class, TypeError),
            stdConstBinding(public::“UninitializedError”, Class, UninitializedError),
            stdConstBinding(public::“URIError”, Class, URIError)},
            archetype: ObjectPrototype, name: “”, initialize: none, sealed: false, internalNamespace: internal⟧
    end proc;
```

16.1 Built-in Namespaces

public: NAMESPACE = **new** NAMESPACE⟦name: “`public`”⟧

internal: NAMESPACE = **new** NAMESPACE⟦name: “`internal`”⟧

16.2 Built-in Attributes

global_explicit: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **true**, enumerable: **false**, dynamic: **false**, category: **none**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_enumerable: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **true**, dynamic: **false**, category: **none**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_dynamic: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **true**, category: **none**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_static: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **false**, category: **static**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_virtual: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **false**, category: **virtual**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_final: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **false**, category: **final**, overrideMod: **none**, prototype: **false**, unused: **false**⟧

global_prototype: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **false**, category: **none**, overrideMod: **none**, prototype: **true**, unused: **false**⟧

global_unused: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE⟦namespaces: {}, explicit: **false**, enumerable: **false**, dynamic: **false**, category: **none**, overrideMod: **none**, prototype: **false**, unused: **true**⟧

```
proc global_override(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note This function does not check phase and therefore can be used in a constant expression.
    overrideMod: OVERRIDEMODIFIER;
    if args = [] then overrideMod ← true
    elsif |args| = 1 then
        arg: OBJECT ← args[0];
        if arg ∉ {true, false, undefined} then throw a TypeError exception end if;
        overrideMod ← arg
    else throw an ArgumentError exception — too many arguments supplied
    end if;
    return COMPOUNDATTRIBUTE⟨⟨namespaces: {}, explicit: false, enumerable: false, dynamic: false,
            category: none, overrideMod: overrideMod, prototype: false, unused: false⟩⟩
end proc;
```

16.3 Built-in Functions

```
proc global_eval(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

```
proc global_parseint(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
    note This function can be used in a constant expression if the arguments can be converted to primitives in constant
            expressions.
    if |args| ∉ {1, 2} then
        throw an ArgumentError exception — at least one and at most two arguments must be supplied
    end if;
    s: STRING ← objectToString(args[0], phase);
    radix: INTEGER ← objectToInteger(defaultArg(args, 1, +zero_f64), phase);
    i: (INTEGER – {0}) ∪ {+zero, –zero, NaN} ← stringPrefixToInteger(s, radix);
    return extendedRationalToFloat64(i)
end proc;
```

```
proc stringPrefixToInteger(s: STRING, radix: INTEGER): (INTEGER – {0}) ∪ {+zero, –zero, NaN}
  r: INTEGER ← radix;
  if r ∉ {0, 2 ... 36} then throw a RangeError exception — radix out of range end if;
  i: INTEGER ← 0;
  while i < |s| and the nonterminal WhiteSpaceOrLineTerminatorChar can expand into [s[i]] do
    i ← i + 1
  end while;
  sign: {–1, 1} ← 1;
  if i < |s| then
    if s[i] = ‘+’ then i ← i + 1 elsif s[i] = ‘-’ then sign ← –1; i ← i + 1 end if
  end if;
  if r ∈ {0, 16} and i + 2 ≤ |s| and s[i ... i + 1] ∈ {“0x”, “0X”} then
    r ← 16;
    i ← i + 2
  end if;
  if r = 0 then r ← 10 end if;
  n: INTEGER ← 0;
  start: INTEGER ← i;
  digit: INTEGEROPT ← 0;
  while i < |s| and digit ≠ none do
    ch: CHAR16 ← s[i];
    if ch ∈ {‘0’ ... ‘9’} then digit ← char16ToInteger(ch) – char16ToInteger(‘0’)
    elsif ch ∈ {‘A’ ... ‘Z’} then
      digit ← char16ToInteger(ch) – char16ToInteger(‘A’) + 10
    elsif ch ∈ {‘a’ ... ‘z’} then
      digit ← char16ToInteger(ch) – char16ToInteger(‘a’) + 10
    else digit ← none
    end if;
    if digit ≠ none and digit ≥ r then digit ← none end if;
    if digit ≠ none then n ← n·r + digit; i ← i + 1 end if
  end while;
  if i = start then return NaN end if;
  if n ≠ 0 then return n·sign
  elsif sign > 0 then return +zero
  else return –zero
  end if
end proc;

proc global_parselong(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): GENERALNUMBER
  note  This function can be used in a constant expression if the arguments can be converted to primitives in constant
        expressions.
  if |args| ∉ {1, 2} then
    throw an ArgumentError exception — at least one and at most two arguments must be supplied
  end if;
  s: STRING ← objectToString(args[0], phase);
  radix: INTEGER ← objectToInteger(defaultArg(args, 1, +zero_f64), phase);
  i: (INTEGER – {0}) ∪ {+zero, –zero, NaN} ← stringPrefixToInteger(s, radix);
  case i of
    {+zero, –zero} do return 0_long;
    INTEGER do return integerToLong(i);
    {NaN} do return NaN_f64
  end case
end proc;
```

```
proc global_parsefloat(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
    note This function can be used in a constant expression if its argument can be converted to a primitive in a constant
        expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    s: STRING  objectToString(args[0], phase);
    Apply the lexer grammar with the start symbol StringDecimalLiteral to the string s. If the grammar can interpret
    neither s nor any prefix of s as an expansion of StringDecimalLiteral, then return $\mathrm{NaN}_{f64}$. Otherwise, let p be the
    longest prefix of s (possibly s itself) such that p is an expansion of StringDecimalLiteral.
    q: EXTENDEDRATIONAL  the value of the action Lex applied to p’s expansion of the nonterminal
    StringDecimalLiteral;
    return extendedRationalToFloat64(q)
end proc;
```

```
proc global_isnan(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): BOOLEAN
    note This function can be used in a constant expression if its argument can be converted to a primitive in a constant
        expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    x: GENERALNUMBER  objectToGeneralNumber(args[0], phase);
    return x  {$\mathrm{NaN}_{f32}$, $\mathrm{NaN}_{f64}$}
end proc;
```

```
proc global_isfinite(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): BOOLEAN
    note This function can be used in a constant expression if its argument can be converted to a primitive in a constant
        expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    x: GENERALNUMBER  objectToGeneralNumber(args[0], phase);
    return x  {$\mathrm{NaN}_{f32}$, $\mathrm{NaN}_{f64}$, $+\infty_{f32}$, $+\infty_{f64}$, $-\infty_{f32}$, $-\infty_{f64}$}
end proc;
```

```
proc global_decodeuri(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

```
proc global_decodeuricomponent(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

```
proc global_encodeuri(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

```
proc global_encodeuricomponent(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

17 Built-in Classes

```
proc dummyCall(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
    Evaluate ???? and ignore its result
end proc;
```

```
proc dummyConstruct(c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
   Evaluate ???? and ignore its result
end proc;

prototypesSealed: BOOLEAN = false;
```

17.1 Object

```
Object: CLASS = new CLASS⟨localBindings: {}, instanceProperties: {}, super: none, prototype: ObjectPrototype,
      complete: true, name: “Object”, typeofString: “object”, dynamic: true, final: false,
      defaultValue: undefined, defaultHint: hintNumber, hasProperty: ordinaryHasProperty,
      bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
      bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
      enumerate: ordinaryEnumerate, call: callObject, construct: constructObject, init: none, is: ordinaryIs,
      coerce: coerceObject⟩

proc callObject(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
   note This function does not check phase and therefore can be used in a constant expression.
   if |args| = 0 then return undefined
   elsif |args| = 1 then return args[0]
   else throw an ArgumentError exception — at most one argument can be supplied
   end if
end proc;

proc constructObject(c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
   note This function does not check phase and therefore can be used in a constant expression.
   if |args| > 1 then
      throw an ArgumentError exception — at most one argument can be supplied
   end if;
   o: OBJECT ← defaultArg(args, 0, undefined);
   if o ∈ {null, undefined} then
      return createSimpleInstance(Object, ObjectPrototype, none, none, none)
   else return o
   end if
end proc;

proc coerceObject(o: OBJECT, c: CLASS): OBJECTOPT
   return o
end proc;

ObjectPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE⟨localBindings: {
      stdConstBinding(public::“constructor”, Class, Object),
      stdFunction(public::“toString”, Object_toString, 0),
      stdFunction(public::“toLocaleString”, Object_toLocaleString, 0),
      stdFunction(public::“valueOf”, Object_valueOf, 0),
      stdFunction(public::“hasOwnProperty”, Object_hasOwnProperty, 1),
      stdFunction(public::“isPrototypeOf”, Object_isPrototypeOf, 1),
      stdFunction(public::“propertyIsEnumerable”, Object_propertyIsEnumerable, 1),
      stdFunction(public::“sealProperty”, Object_sealProperty, 1)},
      archetype: none, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none, env: none⟩

proc Object_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
   note This function does not check phase and therefore can be used in a constant expression.
   note This function ignores any arguments passed to it in args.
   c: CLASS ← objectType(this);
   return “[object ” ⊕ c.name ⊕ “]”
end proc;
```

```
proc Object_toLocaleString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — toLocaleString cannot be called from a constant expression
    end if;
    toStringMethod: OBJECT □ dotRead(this, {public::“toString”}, phase);
    return call(this, toStringMethod, args, phase)
end proc;

proc Object_valueOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note This function does not check phase and therefore can be used in a constant expression.
    note This function ignores any arguments passed to it in args.
    return this
end proc;

proc Object_hasOwnProperty(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): BOOLEAN
    if phase = compile then
        throw a ConstantError exception — hasOwnProperty cannot be called from a constant expression
    end if;
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    return hasProperty(this, args[0], true, phase)
end proc;

proc Object_isPrototypeOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): BOOLEAN
    if phase = compile then
        throw a ConstantError exception — isPrototypeOf cannot be called from a constant expression
    end if;
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    o: OBJECT □ args[0];
    return this □ archetypes(o)
end proc;

proc Object_propertyIsEnumerable(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): BOOLEAN
    if phase = compile then
        throw a ConstantError exception — propertyIsEnumerable cannot be called from a constant expression
    end if;
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    qname: QUALIFIEDNAME □ objectToQualifiedName(args[0], phase);
    c: CLASS □ objectType(this);
    mBase: INSTANCEPROPERTYOPT □ findBaseInstanceProperty(c, {qname}, read);
    if mBase ≠ none then
        m: INSTANCEPROPERTY □ getDerivedInstanceProperty(c, mBase, read);
        if m.enumerable then return true end if
    end if;
    mBase □ findBaseInstanceProperty(c, {qname}, write);
    if mBase ≠ none then
        m: INSTANCEPROPERTY □ getDerivedInstanceProperty(c, mBase, write);
        if m.enumerable then return true end if
    end if;
    if this □ BINDINGOBJECT then return false end if;
    return some b □ this.localBindings satisfies b.qname = qname and b.enumerable
end proc;
```

```
proc Object_sealProperty(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): UNDEFINED
    if phase = compile then
        throw a ConstantError exception — sealProperty cannot be called from a constant expression
    end if;
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT ← defaultArg(args, 0, true);
    if arg = false then Evaluate sealObject(this) and ignore its result
    elsif arg = true then
        Evaluate sealObject(this) and ignore its result;
        Evaluate sealAllLocalProperties(this) and ignore its result
    elsif arg ∈ CHAR16 ∪ STRING then
        if not hasProperty(this, arg, true, phase) then
            throw a ReferenceError exception — property not found
        end if;
        qname: QUALIFIEDNAME ← objectToQualifiedName(arg, phase);
        Evaluate sealLocalProperty(this, qname) and ignore its result
    end if;
    return undefined
end proc;
```

17.2 Never

```
Never: CLASS = new CLASS⟪localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
        complete: true, name: "Never", typeofString: "", dynamic: false, final: true, defaultValue: none,
        hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructNever, init: none, is: ordinaryIs,
        coerce: coerceNever⟫
```

```
proc constructNever(c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    throw a TypeError exception — no coercions to Never are possible
end proc;
```

```
proc coerceNever(o: OBJECT, c: CLASS): {none}
    return none
end proc;
```

17.3 Void

```
Void: CLASS = new CLASS⟪localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
        complete: true, name: "Void", typeofString: "undefined", dynamic: false, final: true,
        defaultValue: undefined, hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead,
        bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
        write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callVoid,
        construct: constructVoid, init: none, is: ordinaryIs, coerce: coerceVoid⟫
```

```
proc callVoid(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): UNDEFINED
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    return undefined
end proc;
```

```
proc constructVoid(c: CLASS, args: OBJECT[], phase: PHASE): UNDEFINED
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    return undefined
end proc;
```

```
proc coerceVoid(o: OBJECT, c: CLASS): {undefined, none}
    if o ∈ NULL ∪ UNDEFINED then return undefined else return none end if
end proc;
```

17.4 Null

```
Null: CLASS = new CLASS⟦localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
        complete: true, name: "Null", typeofString: "object", dynamic: false, final: true, defaultValue: null,
        hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: callNull, construct: constructNull, init: none, is: ordinaryIs,
        coerce: coerceNull⟧
```

```
proc callNull(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): NULL
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    return null
end proc;
```

```
proc constructNull(c: CLASS, args: OBJECT[], phase: PHASE): NULL
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    return null
end proc;
```

```
proc coerceNull(o: OBJECT, c: CLASS): {null, none}
    if o = null then return o else return none end if
end proc;
```

17.5 Boolean

```
Boolean: CLASS = new CLASS⟦localBindings: {}, instanceProperties: {}, super: Object, prototype: BooleanPrototype,
        complete: true, name: "Boolean", typeofString: "boolean", dynamic: false, final: true,
        defaultValue: false, hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead,
        bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
        write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: sameAsConstruct,
        construct: constructBoolean, init: none, is: ordinaryIs, coerce: coerceBoolean⟧
```

```
proc constructBoolean(c: CLASS, args: OBJECT[], phase: PHASE): BOOLEAN
   note  This function does not check phase and therefore can be used in a constant expression.
   if |args| > 1 then
      throw an ArgumentError exception — at most one argument can be supplied
   end if;
   return objectToBoolean(defaultArg(args, 0, false))
end proc;

proc coerceBoolean(o: OBJECT, c: CLASS): BOOLEANOPT
   if o □ BOOLEAN then return o else return none end if
end proc;

BooleanPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
      stdConstBinding(public::“constructor”, Class, Boolean),
      stdFunction(public::“toString”, Boolean_toString, 0),
      stdReserve(public::“valueOf”, ObjectPrototype)},
      archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
      env: none□

proc Boolean_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
   note  This function can be used in a constant expression.
   note  This function ignores any arguments passed to it in args.
   a: BOOLEAN □  objectToBoolean(this);
   return objectToString(a, phase)
end proc;
```

17.6 GeneralNumber

```
GeneralNumber: CLASS = new CLASS□localBindings: {}, instanceProperties: {}, super: Object,
      prototype: GeneralNumberPrototype, complete: true, name: “GeneralNumber”, typeofString: “object”,
      dynamic: false, final: true, defaultValue: NaN_f64, defaultHint: hintNumber,
      hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
      bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
      enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructGeneralNumber, init: none,
      is: ordinaryIs, coerce: coerceGeneralNumber□

proc constructGeneralNumber(c: CLASS, args: OBJECT[], phase: PHASE): GENERALNUMBER
   note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
         expression.
   if |args| = 0 then return +zero_f64
   elsif |args| = 1 then return objectToGeneralNumber(args[0], phase)
   else throw an ArgumentError exception — at most one argument can be supplied
   end if
end proc;

proc coerceGeneralNumber(o: OBJECT, c: CLASS): GENERALNUMBER □ {none}
   if o □ GENERALNUMBER then return o else return none end if
end proc;
```

```
GeneralNumberPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE⟦localBindings: {
        stdConstBinding(public::“constructor”, Class, GeneralNumber),
        stdFunction(public::“toString”, GeneralNumber_toString, 1),
        stdReserve(public::“valueOf”, ObjectPrototype),
        stdFunction(public::“toFixed”, GeneralNumber_toFixed, 1),
        stdFunction(public::“toExponential”, GeneralNumber_toExponential, 1),
        stdFunction(public::“toPrecision”, GeneralNumber_toPrecision, 1)},
        archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
        env: none⟧

proc GeneralNumber_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    note  This function can be used in a constant expression if this and the argument can be converted to primitives in
          constant expressions.
    note  This function is generic and can be applied even if this is not a general number.
    x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
    radix: INTEGER ← objectToInteger(defaultArg(args, 0, $10_{f64}$), phase);
    if radix < 2 or radix > 36 then throw a RangeError exception — bad radix end if;
    if radix = 10 then return generalNumberToString(x)
    else
        return x converted to a string containing a base-radix number in an implementation-defined manner
    end if
end proc;

precisionLimit: INTEGER = an implementation-defined integer not less than 20;

proc GeneralNumber_toFixed(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    note  This function can be used in a constant expression if this and the argument can be converted to primitives in
          constant expressions.
    note  This function is generic and can be applied even if this is not a general number.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
    fractionDigits: INTEGER ← objectToInteger(defaultArg(args, 0, $+zero_{f64}$), phase);
    if fractionDigits < 0 or fractionDigits > precisionLimit then
        throw a RangeError exception
    end if;
    if x ∉ FINITEGENERALNUMBER then return generalNumberToString(x) end if;
    r: RATIONAL ← toRational(x);
    if $|r| \ge 10^{21}$ then return generalNumberToString(x) end if;
    sign: STRING ← “”;
    if r < 0 then sign ← “-”; r ← –r end if;
    n: INTEGER ← $\lfloor r \times 10^{fractionDigits} + 1/2 \rfloor$;
    digits: STRING ← integerToString(n);
    if fractionDigits = 0 then return sign ⊕ digits
    else
        if |digits| ≤ fractionDigits then
            digits ← repeat(‘0’, fractionDigits + 1 – |digits|) ⊕ digits
        end if;
        k: INTEGER ← |digits| – fractionDigits;
        return sign ⊕ digits[0 ... k – 1] ⊕ “.” ⊕ digits[k ...]
    end if
end proc;
```

```
proc GeneralNumber_toExponential(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this and the argument can be converted to primitives in
      constant expressions.
  note This function is generic and can be applied even if this is not a general number.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
  fractionDigits: EXTENDEDINTEGER ← objectToExtendedInteger(defaultArg(args, 0, NaN_f64), phase);
  if fractionDigits ∈ {+∞, –∞} or
      (fractionDigits ≠ NaN and (fractionDigits < 0 or fractionDigits > precisionLimit)) then
    throw a RangeError exception
  end if;
  if x ∉ FINITEGENERALNUMBER then return generalNumberToString(x) end if;
  r: RATIONAL ← toRational(x);
  sign: STRING ← “”;
  if r < 0 then sign ← “-”; r ← –r end if;
  digits: STRING;
  e: INTEGER;
  if fractionDigits ≠ NaN then
    if r = 0 then digits ← repeat(‘0’, fractionDigits + 1); e ← 0
    else
      e ← ⌊log₁₀(r)⌋;
      n: INTEGER ← ⌊r×10^(fractionDigits–e) + 1/2⌋;
      note At this point 10^fractionDigits ≤ n ≤ 10^(fractionDigits+1)
      if n = 10^(fractionDigits+1) then n ← n/10; e ← e + 1 end if;
      digits ← integerToString(n)
    end if;
    note At this point the string digits has exactly fractionDigits + 1 digits
  elsif r = 0 then digits ← “0”; e ← 0
  elsif x ∈ LONG ∪ ULONG then
    digits ← integerToString(r);
    e ← |digits| – 1;
    while digits[|digits| – 1] = ‘0’ do digits ← digits[0 ... |digits| – 2]
    end while
  else
    k: INTEGER;
    s: INTEGER;
    case x of
      NONZEROFINITEFLOAT32 do
        Let e, k, and s be integers such that k ≥ 1, 10^(k–1) ≤ s ≤ 10^k, (s×10^(e+1–k))_f32 = x, and k is as small as possible.
      NONZEROFINITEFLOAT64 do
        Let e, k, and s be integers such that k ≥ 1, 10^(k–1) ≤ s ≤ 10^k, (s×10^(e+1–k))_f64 = x, and k is as small as possible.
    end case;
    note k is the number of digits in the decimal representation of s, s is not divisible by 10, and the least significant
        digit of s is not necessarily uniquely determined by the above criteria.
    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^(e+1–k) is closest in
    value to r; if there are two such possible values of s, choose the one that is even.
    digits ← integerToString(s)
  end if;
  return sign ⊕ exponentialNotationString(digits, e)
end proc;
```

```
proc GeneralNumber_toPrecision(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
   note This function can be used in a constant expression if this and the argument can be converted to primitives in
      constant expressions.
   note This function is generic and can be applied even if this is not a general number.
   if |args| > 1 then
      throw an ArgumentError exception — at most one argument can be supplied
   end if;
   x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
   precision: EXTENDEDINTEGER ← objectToExtendedInteger(defaultArg(args, 0, NaN_f64), phase);
   if precision = NaN then return generalNumberToString(x) end if;
   if precision ∈ {+∞, −∞} or precision < 1 or precision > precisionLimit + 1 then
      throw a RangeError exception
   end if;
   if x ∉ FINITEGENERALNUMBER then return generalNumberToString(x) end if;
   r: RATIONAL ← toRational(x);
   sign: STRING ← “”;
   if r < 0 then sign ← “-”; r ← –r end if;
   digits: STRING;
   e: INTEGER;
   if r = 0 then digits ← repeat(‘0’, precision); e ← 0
   else
      e ← $\lfloor \log_{10}(r) \rfloor$;
      n: INTEGER ← $\lfloor r \times 10^{precision-1-e} + 1/2 \rfloor$;
      note At this point $10^{precision-1} \le n \le 10^{precision}$
      if n = $10^{precision}$ then n ← n/10; e ← e + 1 end if;
      digits ← integerToString(n)
   end if;
   note At this point the string digits has exactly precision digits
   if e < –6 or e ≥ precision then return sign ⊕ exponentialNotationString(digits, e)
   elsif e = precision – 1 then return sign ⊕ digits
   elsif e ≥ 0 then return sign ⊕ digits[0 ... e] ⊕ “.” ⊕ digits[e + 1 ...]
   else return sign ⊕ “0.” ⊕ repeat(‘0’, –(e + 1)) ⊕ digits
   end if
end proc;
```

17.7 long

```
long: CLASS = new CLASS⟦localBindings: {
      stdConstBinding(public::“MAX_VALUE”, ulong, $(2^{63} – 1)_{long}$),
      stdConstBinding(public::“MIN_VALUE”, ulong, $(–2^{63})_{long}$)},
      instanceProperties: {}, super: GeneralNumber, prototype: longPrototype, complete: true, name: “long”,
      typeofString: “long”, dynamic: false, final: true, defaultValue: $0_{long}$, hasProperty: ordinaryHasProperty,
      bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
      bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
      enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructLong, init: none, is: ordinaryIs,
      coerce: coerceLong⟧
```

```
proc constructLong(c: CLASS, args: OBJECT[], phase: PHASE): LONG
    note This function can be used in a constant expression if the argument can be converted to a primitive in a constant
        expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT □ defaultArg(args, 0, +zero_f64);
    i: INTEGER □ objectToInteger(arg, phase);
    if $-2^{63} \le i \le 2^{63} - 1$ then return $i_{long}$
    else throw a RangeError exception — i is out of the LONG range
    end if
end proc;

proc coerceLong(o: OBJECT, c: CLASS): LONG □ {none}
    if o □ GENERALNUMBER then return none end if;
    i: INTEGEROPT □ checkInteger(o);
    if $i \ne$ none and $-2^{63} \le i \le 2^{63} - 1$ then return $i_{long}$
    else throw a RangeError exception — i is out of the LONG range
    end if
end proc;

longPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
        stdConstBinding(public::“constructor”, Class, long),
        stdReserve(public::“toString”, GeneralNumberPrototype),
        stdReserve(public::“valueOf”, GeneralNumberPrototype)},
        archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
        construct: none, env: none□
```

17.8 ulong

```
ulong: CLASS = new CLASS□localBindings: {
        stdConstBinding(public::“MAX_VALUE”, ulong, $(2^{64} - 1)_{ulong}$),
        stdConstBinding(public::“MIN_VALUE”, ulong, $0_{ulong}$)},
        instanceProperties: {}, super: GeneralNumber, prototype: ulongPrototype, complete: true, name: “ulong”,
        typeofString: “ulong”, dynamic: false, final: true, defaultValue: $0_{ulong}$, hasProperty: ordinaryHasProperty,
        bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructULong, init: none, is: ordinaryIs,
        coerce: coerceULong□

proc constructULong(c: CLASS, args: OBJECT[], phase: PHASE): ULONG
    note This function can be used in a constant expression if the argument can be converted to a primitive in a constant
        expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT □ defaultArg(args, 0, +zero_f64);
    i: INTEGER □ objectToInteger(arg, phase);
    if $0 \le i \le 2^{64} - 1$ then return $i_{ulong}$
    else throw a RangeError exception — i is out of the ULONG range
    end if
end proc;
```

```
proc coerceULong(o: Object, c: Class): ULong ∪ {none}
   if o ∉ GeneralNumber then return none end if;
   i: IntegerOpt ← checkInteger(o);
   if i ≠ none and $0 \le i \le 2^{64} - 1$ then return $i_{ulong}$
   else throw a RangeError exception — i is out of the ULong range
   end if
end proc;
```

```
ulongPrototype: SimpleInstance = new SimpleInstance⟦localBindings: {
      stdConstBinding(public::“constructor”, Class, ulong),
      stdReserve(public::“toString”, GeneralNumberPrototype),
      stdReserve(public::“valueOf”, GeneralNumberPrototype)},
      archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
      construct: none, env: none⟧
```

17.9 float

```
float: Class = new Class⟦localBindings: {
      stdConstBinding(public::“MAX_VALUE”, float, $(3.4028235 \times 10^{38})_{f32}$),
      stdConstBinding(public::“MIN_VALUE”, float, $(10^{-45})_{f32}$),
      stdConstBinding(public::“NaN”, float, $NaN_{f32}$),
      stdConstBinding(public::“NEGATIVE_INFINITY”, float, $-\infty_{f32}$),
      stdConstBinding(public::“POSITIVE_INFINITY”, float, $+\infty_{f32}$)},
      instanceProperties: {}, super: GeneralNumber, prototype: floatPrototype, complete: true, name: “float”,
      typeofString: “float”, dynamic: false, final: true, defaultValue: $NaN_{f32}$, hasProperty: ordinaryHasProperty,
      bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
      bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
      enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructFloat, init: none, is: ordinaryIs,
      coerce: coerceFloat⟧
```

```
proc constructFloat(c: Class, args: Object[], phase: Phase): Float32
   note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
         expression.
   if |args| = 0 then return $+zero_{f32}$
   elsif |args| = 1 then return objectToFloat32(args[0], phase)
   else throw an ArgumentError exception — at most one argument can be supplied
   end if
end proc;
```

```
proc coerceFloat(o: Object, c: Class): Float32 ∪ {none}
   if o ∈ GeneralNumber then return toFloat32(o) else return none end if
end proc;
```

```
floatPrototype: SimpleInstance = new SimpleInstance⟦localBindings: {
      stdConstBinding(public::“constructor”, Class, float),
      stdReserve(public::“toString”, GeneralNumberPrototype),
      stdReserve(public::“valueOf”, GeneralNumberPrototype)},
      archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
      construct: none, env: none⟧
```

17.10 Number

```
Number: CLASS = new CLASS⟨⟨localBindings: {
        stdConstBinding(public::“MAX_VALUE”, Number, (1.7976931348623157×10^308)f64),
        stdConstBinding(public::“MIN_VALUE”, Number, (5×10^–324)f64),
        stdConstBinding(public::“NaN”, Number, NaNf64),
        stdConstBinding(public::“NEGATIVE_INFINITY”, Number, –∞f64),
        stdConstBinding(public::“POSITIVE_INFINITY”, Number, +∞f64)},
        instanceProperties: {}, super: GeneralNumber, prototype: NumberPrototype, complete: true,
        name: “Number”, typeofString: “number”, dynamic: false, final: true, defaultValue: NaNf64,
        hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructNumber, init: none, is: ordinaryIs,
        coerce: coerceNumber⟩⟩

proc constructNumber(c: CLASS, args: OBJECT[], phase: PHASE): FLOAT64
    note This function can be used in a constant expression if the argument can be converted to a primitive in a constant
         expression.
    if |args| = 0 then return +zerof64
    elsif |args| = 1 then return objectToFloat64(args[0], phase)
    else throw an ArgumentError exception — at most one argument can be supplied
    end if
end proc;

proc coerceNumber(o: OBJECT, c: CLASS): FLOAT64 ∪ {none}
    if o ∈ GENERALNUMBER then return toFloat64(o) else return none end if
end proc;

NumberPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE⟨⟨localBindings: {
        stdConstBinding(public::“constructor”, Class, Number),
        stdReserve(public::“toString”, GeneralNumberPrototype),
        stdReserve(public::“valueOf”, GeneralNumberPrototype)},
        archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
        construct: none, env: none⟩⟩
```

```
proc makeBuiltInIntegerClass(name: STRING, low: INTEGER, high: INTEGER): CLASS
    proc construct(c: CLASS, args: OBJECT[], phase: PHASE): FLOAT64
        note  This function can be used in a constant expression if the argument can be converted to a primitive in a
              constant expression.
        if |args| > 1 then
            throw an ArgumentError exception — at most one argument can be supplied
        end if;
        arg: OBJECT □ defaultArg(args, 0, $+\mathbf{zero}_{f64}$);
        x: FLOAT64 □ objectToFloat64(arg, phase);
        i: INTEGEROPT □ checkInteger(x);
        if $i \neq$ none and $low \leq i \leq high$ then
            note  $-\mathbf{zero}_{f64}$ is coerced to $+\mathbf{zero}_{f64}$.
            return $i_{f64}$
        end if;
        throw a RangeError exception
    end proc;
    proc is(o: OBJECT, c: CLASS): BOOLEAN
        if o □ FLOAT64 then return false end if;
        i: INTEGEROPT □ checkInteger(o);
        return $i \neq$ none and $low \leq i \leq high$
    end proc;
    proc coerce(o: OBJECT, c: CLASS): FLOAT64 □ {none}
        if o □ GENERALNUMBER then return none end if;
        i: INTEGEROPT □ checkInteger(o);
        if $i \neq$ none and $low \leq i \leq high$ then
            note  $-\mathbf{zero}_{f32}$, $+\mathbf{zero}_{f32}$, and $-\mathbf{zero}_{f64}$ are all coerced to $+\mathbf{zero}_{f64}$.
            return $i_{f64}$
        end if;
        throw a RangeError exception
    end proc;
    return new CLASS□localBindings: {
            stdConstBinding(public::"MAX_VALUE", Number, $high_{f64}$),
            stdConstBinding(public::"MIN_VALUE", Number, $low_{f64}$)},
            instanceProperties: {}, super: Number, prototype: Number.prototype, complete: true, name: name,
            typeofString: "number", dynamic: false, final: true, defaultValue: $+\mathbf{zero}_{f64}$,
            hasProperty: Number.hasProperty, bracketRead: Number.bracketRead,
            bracketWrite: Number.bracketWrite, bracketDelete: Number.bracketDelete, read: Number.read,
            write: Number.write, delete: Number.delete, enumerate: Number.enumerate, call: sameAsConstruct,
            construct: construct, init: none, is: is, coerce: coerce□
end proc;

sbyte: CLASS = makeBuiltInIntegerClass("sbyte", –128, 127);

byte: CLASS = makeBuiltInIntegerClass("byte", 0, 255);

short: CLASS = makeBuiltInIntegerClass("short", –32768, 32767);

ushort: CLASS = makeBuiltInIntegerClass("ushort", 0, 65535);

int: CLASS = makeBuiltInIntegerClass("int", –2147483648, 2147483647);

uint: CLASS = makeBuiltInIntegerClass("uint", 0, 4294967295);
```

17.11 char

```
char: CLASS = new CLASS□localBindings: {stdFunction(public::“fromCharCode”, char_fromCharCode, 1)},
        instanceProperties: {}, super: Object, prototype: charPrototype, complete: true, name: “char”,
        typeofString: “char”, dynamic: false, final: true, defaultValue: ‘«NUL»’, hasProperty: ordinaryHasProperty,
        bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructChar, init: none, is: ordinaryIs,
        coerce: coerceChar□

proc callChar(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): CHAR16
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
          expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    s: STRING □ objectToString(args[0], phase);
    if |s| ≠ 1 then throw a RangeError exception — only one character may be given end if;
    return s[0]
end proc;

proc constructChar(c: CLASS, args: OBJECT[], phase: PHASE): CHAR16
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
          expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT □ defaultArg(args, 0, undefined);
    if arg = undefined then return ‘«NUL»’
    elsif arg □ CHAR16 then return arg
    else
        s: STRING □ objectToString(args[0], phase);
        if |s| ≠ 1 then throw a RangeError exception — only one character may be given
        end if;
        return s[0]
    end if
end proc;

proc coerceChar(o: OBJECT, c: CLASS): CHAR16 □ {none}
    if o □ CHAR16 then return o else return none end if
end proc;

proc char_fromCharCode(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
          expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    i: INTEGER □ objectToInteger(args[0], phase);
    if 0 ≤ i ≤ 0xFFFF then return integerToChar16(i)
    else throw a RangeError exception — character code out of range
    end if
end proc;
```

```
charPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
    stdConstBinding(public::“constructor”, Class, char),
    stdReserve(public::“toString”, StringPrototype),
    stdReserve(public::“valueOf”, StringPrototype)},
    archetype: StringPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
    env: none□
```

17.12 String

```
String: CLASS = new CLASS□localBindings: {stdFunction(public::“fromCharCode”, String_fromCharCode, 1)},
    instanceProperties: {
    new INSTANCEGETTER□multiname: {public::“length”}, final: true, enumerable: false, call: String_length□},
    super: Object, prototype: StringPrototype, complete: true, name: “String”, typeofString: “string”,
    dynamic: false, final: true, defaultValue: null, hasProperty: stringHasProperty,
    bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
    bracketDelete: ordinaryBracketDelete, read: readString, write: ordinaryWrite, delete: ordinaryDelete,
    enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructString, init: none, is: ordinaryIs,
    coerce: coerceString□
```

```
proc stringHasProperty(o: OBJECT, c: CLASS, property: OBJECT, flat: BOOLEAN, phase: PHASE): BOOLEAN
  note  o □ STRING because stringHasProperty is only called on instances of class String.
  qname: QUALIFIEDNAME □ objectToQualifiedName(property, phase);
  i: INTEGEROPT □ multinameToUnsignedInteger({qname});
  if i ≠ none then return i < |o|
  else
    return findBaseInstanceProperty(c, {qname}, read) ≠ none or
        findBaseInstanceProperty(c, {qname}, write) ≠ none or
        findArchetypeProperty(o, {qname}, read, flat) ≠ none or
        findArchetypeProperty(o, {qname}, write, flat) ≠ none
  end if
end proc;
```

```
proc readString(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT,
    undefinedIfMissing: BOOLEAN, phase: PHASE): OBJECTOPT
  note  o □ STRING because readString is only called on instances of class String.
  if limit = String then
    i: INTEGEROPT □ multinameToUnsignedInteger(multiname);
    if i ≠ none then
      if i < |o| then return o[i]
      elsif undefinedIfMissing then return undefined
      else return none
      end if
    end if
  end if;
  return ordinaryRead(o, limit, multiname, env, undefinedIfMissing, phase)
end proc;
```

```
proc constructString(c: CLASS, args: OBJECT[], phase: PHASE): STRING
  note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
        expression.
  if |args| = 0 then return “”
  elsif |args| = 1 then return objectToString(args[0], phase)
  else throw an ArgumentError exception — at most one argument can be supplied
  end if
end proc;
```

```
proc coerceString(o: OBJECT, c: CLASS): STRING ∪ NULL ∪ {none}
    if o ∈ NULL ∪ STRING then return o
    elsif o ∈ CHAR16 then return [o]
    else return none
    end if
end proc;

proc String_length(this: OBJECT, phase: PHASE): OBJECT
    note  this ∈ STRING because this getter cannot be extracted from the String class.
    length: INTEGER ← |this|;
    return length_f64
end proc;

proc String_fromCharCode(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note  This function can be used in a constant expression if the arguments can be converted to primitives in constant
          expressions.
    s: STRING ← “”;
    for each arg ∈ args do
        i: INTEGER ← objectToInteger(arg, phase);
        if 0 ≤ i ≤ 0x10FFFF then s ← s ⊕ integerToUTF16(i)
        else throw a RangeError exception — character code out of range
        end if
    end for each;
    return s
end proc;

StringPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE⟦localBindings: {
        stdConstBinding(public::“constructor”, Class, String),
        stdFunction(public::“toString”, String_toString, 0),
        stdReserve(public::“valueOf”, ObjectPrototype),
        stdFunction(public::“charAt”, String_charAt, 1),
        stdFunction(public::“charCodeAt”, String_charCodeAt, 1),
        stdFunction(public::“concat”, String_concat, 1),
        stdFunction(public::“indexOf”, String_indexOf, 1),
        stdFunction(public::“lastIndexOf”, String_lastIndexOf, 1),
        stdFunction(public::“localeCompare”, String_localeCompare, 1),
        stdFunction(public::“match”, String_match, 1),
        stdFunction(public::“replace”, String_replace, 1),
        stdFunction(public::“search”, String_search, 1),
        stdFunction(public::“slice”, String_slice, 2),
        stdFunction(public::“split”, String_split, 2),
        stdFunction(public::“substring”, String_substring, 2),
        stdFunction(public::“toLowerCase”, String_toLowerCase, 0),
        stdFunction(public::“toLocaleLowerCase”, String_toLocaleLowerCase, 0),
        stdFunction(public::“toUpperCase”, String_toUpperCase, 0),
        stdFunction(public::“toLocaleUpperCase”, String_toLocaleUpperCase, 0)},
        archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
        env: none⟧

proc String_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    note  This function can be used in a constant expression if this can be converted to a primitive in a constant expression.
    note  This function is generic and can be applied even if this is not a string.
    note  This function ignores any arguments passed to it in args.
    return objectToString(this, phase)
end proc;
```

```
proc String_charAt(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this and the argument can be converted to primitives in
       constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  position: EXTENDEDINTEGER ← objectToExtendedInteger(defaultArg(args, 0, $+\mathbf{zero}_{f64}$), phase);
  if position = NaN then throw a RangeError exception
  elsif position ∉ {+∞, –∞} and $0 \le position < |s|$ then return [s[position]]
  else return “”
  end if
end proc;

proc String_charCodeAt(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
  note This function can be used in a constant expression if this and the argument can be converted to primitives in
       constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  position: EXTENDEDINTEGER ← objectToExtendedInteger(defaultArg(args, 0, $+\mathbf{zero}_{f64}$), phase);
  if position = NaN then throw a RangeError exception
  elsif position ∉ {+∞, –∞} and $0 \le position < |s|$ then
    return $(char16ToInteger(s[position]))_{f64}$
  else return $\mathbf{NaN}_{f64}$
  end if
end proc;

proc String_concat(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this and the argument can be converted to primitives in
       constant expressions.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  for each arg ∈ args do s ← s ⊕ objectToString(arg, phase) end for each;
  return s
end proc;

proc String_indexOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
  note This function can be used in a constant expression if this and the arguments can be converted to primitives in
       constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| ∉ {1, 2} then
    throw an ArgumentError exception — at least one and at most two arguments must be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  pattern: STRING ← objectToString(args[0], phase);
  arg: OBJECT ← defaultArg(args, 1, $+\mathbf{zero}_{f64}$);
  position: INTEGER ← pinExtendedInteger(objectToExtendedInteger(arg, phase), |s|, false);
  while $position + |pattern| \le |s|$ do
    if s[position ... position + |pattern| – 1] = pattern then return $position_{f64}$
    end if;
    position ← position + 1
  end while;
  return $(–1)_{f64}$
end proc;
```

```
proc String_lastIndexOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
    note This function can be used in a constant expression if this and the arguments can be converted to primitives in
         constant expressions.
    note This function is generic and can be applied even if this is not a string.
    if |args| ∉ {1, 2} then
        throw an ArgumentError exception — at least one and at most two arguments must be supplied
    end if;
    s: STRING ← objectToString(this, phase);
    pattern: STRING ← objectToString(args[0], phase);
    arg: OBJECT ← defaultArg(args, 1, +∞_f64);
    position: INTEGER ← pinExtendedInteger(objectToExtendedInteger(arg, phase), |s|, false);
    if position + |pattern| > |s| then position ← |s| – |pattern| end if;
    while position ≥ 0 do
        if s[position ... position + |pattern| – 1] = pattern then return position_f64
        end if;
        position ← position – 1
    end while;
    return (–1)_f64
end proc;
```

```
proc String_localeCompare(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
    note This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — localeCompare cannot be called from a constant expression
    end if;
    if |args| < 1 then
        throw an ArgumentError exception — at least one argument must be supplied
    end if;
    s1: STRING ← objectToString(this, phase);
    s2: STRING ← objectToString(args[0], phase);
    Let result: OBJECT be a value of type Number that is the result of a locale-sensitive string comparison of s1 and s2. The
    two strings are compared in an implementation-defined fashion. The result is intended to order strings in the sort order
    specified by the system default locale, and will be negative, zero, or positive, depending on whether s1 comes before s2
    in the sort order, they are equal, or s1 comes after s2 in the sort order, respectively. The result shall not be NaN_f64. The
    comparison shall be a consistent comparison function on the set of all strings.
    return result
end proc;
```

```
proc String_match(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — match cannot be called from a constant expression
    end if;
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    s: STRING ← objectToString(this, phase);
    Evaluate ???? and ignore its result
end proc;
```

```
proc String_replace(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
    note  This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — replace cannot be called from a constant expression
    end if;
    if |args| $\neq$ 2 then
        throw an ArgumentError exception — exactly two arguments must be supplied
    end if;
    s: String □  objectToString(this, phase);
    Evaluate ???? and ignore its result
end proc;

proc String_search(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
    note  This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — search cannot be called from a constant expression
    end if;
    if |args| $\neq$ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    s: String □  objectToString(this, phase);
    Evaluate ???? and ignore its result
end proc;

proc String_slice(this: Object, f: SimpleInstance, args: Object[], phase: Phase): String
    note  This function can be used in a constant expression if this and the arguments can be converted to primitives in
          constant expressions.
    note  This function is generic and can be applied even if this is not a string.
    if |args| > 2 then
        throw an ArgumentError exception — at most two arguments can be supplied
    end if;
    s: String □  objectToString(this, phase);
    startArg: Object □  defaultArg(args, 0, $+\mathbf{zero}_{f64}$);
    endArg: Object □  defaultArg(args, 1, $+\infty_{f64}$);
    start: Integer □  pinExtendedInteger(objectToExtendedInteger(startArg, phase), |s|, true);
    end: Integer □  pinExtendedInteger(objectToExtendedInteger(endArg, phase), |s|, true);
    if start < end then return s[start ... end – 1] else return “” end if
end proc;

proc String_split(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
    note  This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — split cannot be called from a constant expression
    end if;
    if |args| > 2 then
        throw an ArgumentError exception — at most two arguments can be supplied
    end if;
    s: String □  objectToString(this, phase);
    Evaluate ???? and ignore its result
end proc;
```

```
proc String_substring(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this and the arguments can be converted to primitives in
       constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  startArg: OBJECT ← defaultArg(args, 0, +zero_f64);
  endArg: OBJECT ← defaultArg(args, 1, +∞_f64);
  start: INTEGER ← pinExtendedInteger(objectToExtendedInteger(startArg, phase), |s|, false);
  end: INTEGER ← pinExtendedInteger(objectToExtendedInteger(endArg, phase), |s|, false);
  if start ≤ end then return s[start ... end – 1]
  else return s[end ... start – 1]
  end if
end proc;

proc String_toLowerCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this can be converted to a primitive in a constant expression.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  s32: CHAR21[] ← stringToUTF32(s);
  r: STRING ← “”;
  for each ch ∈ s32 do r ← r ⊕ charToLowerFull(ch) end for each;
  return r
end proc;

proc String_toLocaleLowerCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — toLocaleLowerCase cannot be called from a constant expression
  end if;
  s: STRING ← objectToString(this, phase);
  s32: CHAR21[] ← stringToUTF32(s);
  r: STRING ← “”;
  for each ch ∈ s32 do r ← r ⊕ charToLowerLocalized(ch) end for each;
  return r
end proc;

proc String_toUpperCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression if this can be converted to a primitive in a constant expression.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  s32: CHAR21[] ← stringToUTF32(s);
  r: STRING ← “”;
  for each ch ∈ s32 do r ← r ⊕ charToUpperFull(ch) end for each;
  return r
end proc;
```

```
proc String_toLocaleUpperCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    note This function is generic and can be applied even if this is not a string.
    if phase = compile then
        throw a ConstantError exception — toLocaleUpperCase cannot be called from a constant expression
    end if;
    s: STRING ← objectToString(this, phase);
    s32: CHAR21[] ← stringToUTF32(s);
    r: STRING ← “”;
    for each ch ∈ s32 do r ← r ⊕ charToUpperLocalized(ch) end for each;
    return r
end proc;
```

17.13 Array

```
Array: CLASS = new CLASS⟪localBindings: {}, instanceProperties: {
        new INSTANCEVARIABLE⟪multiname: {arrayPrivate::“length”}, final: true, enumerable: false, type: Number,
        defaultValue: +zero_f64, immutable: false⟫,
        new INSTANCEGETTER⟪multiname: {public::“length”}, final: true, enumerable: false, call: Array_getLength⟫
        new INSTANCESETTER⟪multiname: {public::“length”}, final: true, enumerable: false, call: Array_setLength⟫},
        super: Object, prototype: ArrayPrototype, complete: true, name: “Array”, typeofString: “object”,
        privateNamespace: arrayPrivate, dynamic: true, final: true, defaultValue: null, defaultHint: hintNumber,
        hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: writeArray, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: ordinaryConstruct, init: initArray,
        is: ordinaryIs, coerce: ordinaryCoerce⟫
```

arrayLimit: INTEGER = an implementation-defined integer value between $2^{32} – 1$ and 2^{53} inclusive;

arrayPrivate: NAMESPACE = **new** NAMESPACE⟪name: “private”⟫

```
proc writeArray(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, newValue: OBJECT,
        createIfMissing: BOOLEAN, phase: {run}): {none, ok}
    result: {none, ok} ← ordinaryWrite(o, limit, multiname, env, newValue, createIfMissing, phase);
    if result = ok then
        i: INTEGEROPT ← multinameToUnsignedInteger(multiname);
        if i ≠ none then
            if i ≥ arrayLimit then throw a RangeError exception — array index out of range
            end if;
            length: INTEGER ← readArrayPrivateLength(o, phase);
            if i ≥ length then
                length ← i + 1;
                Evaluate writeArrayPrivateLength(o, length, phase) and ignore its result
            end if
        end if
    end if;
    return result
end proc;
```

readArrayPrivateLength(*array*, *phase*) returns an `Array`’s private length. See also *readLength*, which can work on non-`Array` objects.

```
proc readArrayPrivateLength(array: OBJECT, phase: PHASE): INTEGER
    length: FLOAT64 ← readInstanceSlot(array, arrayPrivate::“length”, phase);
    note length ∉ {NaN_f64, +∞_f64, –∞_f64};
    n: RATIONAL ← toRational(length);
    note n ∈ INTEGER and 0 ≤ n ≤ arrayLimit;
    return n
end proc;
```

writeArrayPrivateLength(*array*, *length*, *phase*) sets an `Array`'s private length to *length* after ensuring that *length* is between 0 and *arrayLimit* inclusive. See also *writeLength*, which can work on non-`Array` objects.

```
proc writeArrayPrivateLength(array: OBJECT, length: INTEGER, phase: {run})
    if length < 0 or length > arrayLimit then
        throw a RangeError exception — array length out of range
    end if;
    Evaluate dotWrite(array, {arrayPrivate::“length”}, $length_{f64}$, phase) and ignore its result
end proc;

proc multinameToUnsignedInteger(multiname: MULTINAME): INTEGEROPT
    if |multiname| ≠ 1 then return none end if;
    qname: QUALIFIEDNAME □ the one element of multiname;
    if qname.namespace ≠ public then return none end if;
    name: STRING □ qname.id;
    if name ≠ [] then
        if name = “0” then return 0
        elsif name[0] ≠ ‘0’ and (every ch □ name satisfies ch □ {‘0’ ... ‘9’}) then
            return stringToExtendedInteger(name)
        end if
    end if;
    return none
end proc;

proc initArray(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
    if |args| = 1 then
        arg: OBJECT □ args[0];
        if arg □ GENERALNUMBER then
            length: INTEGEROPT □ checkInteger(arg);
            if length = none then
                throw a RangeError exception — array length must be an integer
            end if;
            Evaluate writeArrayPrivateLength(this, length, phase) and ignore its result;
            return
        end if
    end if;
    i: INTEGER □ 0;
    for each arg □ args do
        Evaluate indexWrite(this, i, arg, phase) and ignore its result;
        i □ i + 1
    end for each;
    note The call to indexWrite above also set the array’s length to i.
end proc;

proc Array_getLength(this: OBJECT, phase: PHASE): FLOAT64
    note is(this, Array) because this getter cannot be extracted from the Array class.
    note An array’s length is mutable, so reading it will throw ConstantError when phase = compile.
    return readInstanceSlot(this, arrayPrivate::“length”, phase)
end proc;
```

```
proc Array_setLength(this: OBJECT, length: OBJECT, phase: PHASE)
    note  is(this, Array) because this setter cannot be extracted from the Array class.
    if phase = compile then
        throw a ConstantError exception — an array's length cannot be set from a constant expression
    end if;
    newLength: INTEGEROPT □  checkInteger(objectToGeneralNumber(length, phase));
    if newLength = none or newLength < 0 or newLength > arrayLimit then
        throw a RangeError exception — array length out of range or not an integer
    end if;
    oldLength: INTEGER □  readArrayPrivateLength(this, phase);
    if newLength < oldLength then
        note  Delete all indexed properties greater than or equal to the new length
        proc qnameInDeletedRange(qname: QUALIFIEDNAME): BOOLEAN
            i: INTEGEROPT □  multinameToUnsignedInteger({qname});
            return i ≠ none and newLength ≤ i < oldLength
        end proc;
        this.localBindings □  {b | □b □ this.localBindings such that not qnameInDeletedRange(b.qname)}
    end if;
    Evaluate writeArrayPrivateLength(this, newLength, phase) and ignore its result
end proc;

ArrayPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
        stdConstBinding(public::"constructor", Class, Array),
        stdFunction(public::"toString", Array_toString, 0),
        stdFunction(public::"toLocaleString", Array_toLocaleString, 0),
        stdFunction(public::"concat", Array_concat, 1),
        stdFunction(public::"join", Array_join, 1),
        stdFunction(public::"pop", Array_pop, 0),
        stdFunction(public::"push", Array_push, 1),
        stdFunction(public::"reverse", Array_reverse, 0),
        stdFunction(public::"shift", Array_shift, 0),
        stdFunction(public::"slice", Array_slice, 2),
        stdFunction(public::"sort", Array_sort, 1),
        stdFunction(public::"splice", Array_splice, 2),
        stdFunction(public::"unshift", Array_unshift, 1)},
        archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
        env: none□

proc Array_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    if phase = compile then
        throw a ConstantError exception — toString cannot be called on an Array from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    note  This function ignores any arguments passed to it in args.
    return internalJoin(this, ",", phase)
end proc;
```

```
proc Array_toLocaleString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    if phase = compile then
        throw a ConstantError exception — toLocaleString cannot be called on an Array from a constant
            expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    note  This function passes any arguments passed to it in args to toLocaleString applied to the elements of the
        array.
    separator: STRING ← the list-separator string appropriate for the host’s current locale, derived in an implementation-
        defined way;
    length: INTEGER ← readLength(this, phase);
    result: STRING ← “”;
    i: INTEGER ← 0;
    while i ≠ length do
        elt: OBJECTOPT ← indexRead(this, i, phase);
        if elt ∉ {undefined, null, none} then
            toLocaleStringMethod: OBJECT ← dotRead(elt, {public::“toLocaleString”}, phase);
            s: OBJECT ← call(elt, toLocaleStringMethod, args, phase);
            if s ∉ CHAR16 ∪ STRING then
                throw a TypeError exception — toLocaleString should return a string
            end if;
            result ← result ⊕ toString(s)
        end if;
        i ← i + 1;
        if i ≠ length then result ← result ⊕ separator end if
    end while;
    return result
end proc;

proc Array_concat(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — concat cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    constituents: OBJECT[] ← [this] ⊕ args;
    array: OBJECT ← construct(Array, [], phase);
    i: INTEGER ← 0;
    for each o ∈ constituents do
        if is(o, Array) then
            oLength: INTEGER ← readLength(o, phase);
            k: INTEGER ← 0;
            while k ≠ oLength do
                elt: OBJECTOPT ← indexRead(o, k, phase);
                if elt ≠ none then
                    Evaluate indexWrite(array, i, elt, phase) and ignore its result
                end if;
                k ← k + 1;
                i ← i + 1
            end while
        else Evaluate indexWrite(array, i, o, phase) and ignore its result; i ← i + 1
        end if
    end for each;
    Evaluate writeArrayPrivateLength(array, i, phase) and ignore its result;
    return array
end proc;
```

```
proc Array_join(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
   if phase = compile then
      throw a ConstantError exception — join cannot be called from a constant expression
   end if;
   note  This function is generic and can be applied even if this is not an Array.
   if |args| > 1 then
      throw an ArgumentError exception — at most one argument can be supplied
   end if;
   arg: OBJECT □  defaultArg(args, 0, undefined);
   separator: STRING □  “,”;
   if arg ≠ undefined then separator □  objectToString(arg, phase) end if;
   return internalJoin(this, separator, phase)
end proc;

proc internalJoin(this: OBJECT, separator: STRING, phase: {run}): STRING
   length: INTEGER □  readLength(this, phase);
   result: STRING □  “”;
   i: INTEGER □  0;
   while i ≠ length do
      elt: OBJECTOPT □  indexRead(this, i, phase);
      if elt □  {undefined, null, none} then
         result □  result ⊕ objectToString(elt, phase)
      end if;
      i □  i + 1;
      if i ≠ length then result □  result ⊕ separator end if
   end while;
   return result
end proc;

proc Array_pop(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
   if phase = compile then
      throw a ConstantError exception — pop cannot be called from a constant expression
   end if;
   note  This function is generic and can be applied even if this is not an Array.
   if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
   end if;
   length: INTEGER □  readLength(this, phase);
   result: OBJECT □  undefined;
   if length ≠ 0 then
      length □  length – 1;
      elt: OBJECTOPT □  indexRead(this, length, phase);
      if elt ≠ none then
         result □  elt;
         Evaluate indexWrite(this, length, none, phase) and ignore its result
      end if
   end if;
   Evaluate writeLength(this, length, phase) and ignore its result;
   return result
end proc;
```

```
proc Array_push(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — push cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    length: INTEGER ← readLength(this, phase);
    for each arg ∈ args do
        Evaluate indexWrite(this, length, arg, phase) and ignore its result;
        length ← length + 1
    end for each;
    Evaluate writeLength(this, length, phase) and ignore its result;
    return $length_{f64}$
end proc;

proc Array_reverse(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — reverse cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    length: INTEGER ← readLength(this, phase);
    lo: INTEGER ← 0;
    hi: INTEGER ← length – 1;
    while lo < hi do
        loElt: OBJECTOPT ← indexRead(this, lo, phase);
        hiElt: OBJECTOPT ← indexRead(this, hi, phase);
        Evaluate indexWrite(this, lo, hiElt, phase) and ignore its result;
        Evaluate indexWrite(this, hi, loElt, phase) and ignore its result;
        lo ← lo + 1;
        hi ← hi – 1
    end while;
    return this
end proc;
```

```
proc Array_shift(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — shift cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    length: INTEGER ← readLength(this, phase);
    result: OBJECT ← undefined;
    if length ≠ 0 then
        elt: OBJECTOPT ← indexRead(this, 0, phase);
        if elt ≠ none then result ← elt end if;
        i: INTEGER ← 1;
        while i ≠ length do
            elt ← indexRead(this, i, phase);
            Evaluate indexWrite(this, i – 1, elt, phase) and ignore its result;
            i ← i + 1
        end while;
        length ← length – 1;
        Evaluate indexWrite(this, length, none, phase) and ignore its result
    end if;
    Evaluate writeLength(this, length, phase) and ignore its result;
    return result
end proc;

proc Array_slice(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — slice cannot be called on an Array from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    if |args| > 2 then
        throw an ArgumentError exception — at most two arguments can be supplied
    end if;
    length: INTEGER ← readLength(this, phase);
    startArg: OBJECT ← defaultArg(args, 0, +zero_f64);
    endArg: OBJECT ← defaultArg(args, 1, +∞_f64);
    start: INTEGER ← pinExtendedInteger(objectToExtendedInteger(startArg, phase), length, true);
    end: INTEGER ← pinExtendedInteger(objectToExtendedInteger(endArg, phase), length, true);
    return makeArraySlice(this, start, end, phase)
end proc;

proc makeArraySlice(array: OBJECT, start: INTEGER, end: INTEGER, phase: {run}): OBJECT
    slice: OBJECT ← construct(Array, [], phase);
    i: INTEGER ← start;
    j: INTEGER ← 0;
    while i < end do
        elt: OBJECTOPT ← indexRead(array, i, phase);
        Evaluate indexWrite(slice, j, elt, phase) and ignore its result;
        i ← i + 1;
        j ← j + 1
    end while;
    Evaluate writeLength(slice, j, phase) and ignore its result;
    return slice
end proc;
```

```
proc Array_sort(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — sort cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    Evaluate ???? and ignore its result
end proc;

proc Array_splice(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    if phase = compile then
        throw a ConstantError exception — splice cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    if |args| < 2 then
        throw an ArgumentError exception — at least two arguments must be supplied
    end if;
    length: INTEGER □ readLength(this, phase);
    startArg: OBJECT □ defaultArg(args, 0, +zero_f64);
    deleteCountArg: OBJECT □ defaultArg(args, 1, +zero_f64);
    start: INTEGER □ pinExtendedInteger(objectToExtendedInteger(startArg, phase), length, true);
    deleteCount: INTEGER □ pinExtendedInteger(objectToExtendedInteger(deleteCountArg, phase), length – start, false);
    deletedSlice: OBJECT □ makeArraySlice(this, start, start + deleteCount, phase);
    newElts: OBJECT[] □ args[2 ...];
    newEltCount: INTEGER □ |newElts|;
    countDiff: INTEGER □ newEltCount – deleteCount;
    i: INTEGER;
    if countDiff < 0 then
        i □ start + deleteCount;
        while i ≠ length do
            elt: OBJECTOPT □ indexRead(this, i, phase);
            Evaluate indexWrite(this, i + countDiff, elt, phase) and ignore its result;
            i □ i + 1
        end while;
        i □ 0;
        while i ≠ countDiff do
            i □ i – 1;
            Evaluate indexWrite(this, length + i, none, phase) and ignore its result
        end while
    elsif countDiff > 0 then
        i □ length;
        while i ≠ start + deleteCount do
            i □ i – 1;
            elt: OBJECTOPT □ indexRead(this, i, phase);
            Evaluate indexWrite(this, i + countDiff, elt, phase) and ignore its result
        end while
    end if;
    Evaluate writeLength(this, length + countDiff, phase) and ignore its result;
    i □ start;
    for each arg □ newElts do
        Evaluate indexWrite(this, i, arg, phase) and ignore its result;
        i □ i + 1
    end for each;
    return deletedSlice
end proc;
```

```
proc Array_unshift(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64
    if phase = compile then
        throw a ConstantError exception — unshift cannot be called from a constant expression
    end if;
    note  This function is generic and can be applied even if this is not an Array.
    i: INTEGER ← readLength(this, phase);
    nArgs: INTEGER ← |args|;
    newLength: INTEGER ← nArgs + i;
    if nArgs = 0 then
        At the implementation’s discretion, either do nothing or return newLength_f64
    end if;
    Evaluate writeLength(this, newLength, phase) and ignore its result;
    while i ≠ 0 do
        i ← i – 1;
        elt: OBJECTOPT ← indexRead(this, i, phase);
        Evaluate indexWrite(this, i + nArgs, elt, phase) and ignore its result
    end while;
    for each arg ∈ args do
        Evaluate indexWrite(this, i, arg, phase) and ignore its result;
        i ← i + 1
    end for each;
    return newLength_f64
end proc;
```

17.14 Namespace

Namespace: CLASS = **new** CLASS⟨localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *NamespacePrototype*, complete: **true**, name: “`Namespace`”, typeofString: “`namespace`”, dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *ordinaryCall*, construct: *constructNamespace*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*⟩

```
proc constructNamespace(c: CLASS, args: OBJECT[], phase: PHASE): NAMESPACE
  note  This function can be used in a constant expression if its argument is a string.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT □ defaultArg(args, 0, undefined);
  if arg □ NULL □ UNDEFINED then
    if phase = compile then
      throw a ConstantError exception — a constant expression cannot construct new anonymous namespaces
    end if;
    return new NAMESPACE□name: “anonymous”□
  elsif arg □ CHAR16 □ STRING then
    name: STRING □ toString(arg);
    if name = “” then return public
    elsif some ns □ namedNamespaces satisfies ns.name = name then return ns
    else
      ns2: NAMESPACE □ new NAMESPACE□name: name□
      namedNamespaces □ namedNamespaces □ {ns2};
      return ns2
    end if
  else throw a TypeError exception
  end if
end proc;
```

```
namedNamespaces: NAMESPACE{} □ {};
```

```
NamespacePrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
    stdFunction(public::“toString”, Namespace_toString, 0),
    stdReserve(public::“valueOf”, ObjectPrototype)},
    archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
    env: none□
```

```
proc Namespace_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note  This function does not check phase and therefore can be used in a constant expression.
  note  This function ignores any arguments passed to it in args.
  if this □ NAMESPACE then throw a TypeError exception end if;
  return this.name
end proc;
```

17.15 Attribute

```
Attribute: CLASS = new CLASS□localBindings: {}, instanceProperties: {}, super: Object, prototype: ObjectPrototype,
    complete: true, name: “Attribute”, typeofString: “object”, dynamic: false, final: true,
    defaultValue: null, defaultHint: hintString, hasProperty: ordinaryHasProperty,
    bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
    bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
    enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: none, is: ordinaryIs,
    coerce: ordinaryCoerce□
```

17.16 Date

Date: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *DatePrototype*, complete: **true**, name: “`Date`”, typeofString: “`object`”, dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

DatePrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE□localBindings: {}, archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**□

17.17 RegExp

RegExp: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *RegExpPrototype*, complete: **true**, name: “`RegExp`”, typeofString: “`object`”, dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintNumber**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

RegExpPrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE□localBindings: {}, archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**□

17.18 Class

Class: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {*classPrototypeGetter*}, super: *Object*, prototype: *ClassPrototype*, complete: **true**, name: “`Class`”, typeofString: “`function`”, dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

classPrototypeGetter: INSTANCEGETTER = **new** INSTANCEGETTER□multiname: {*public*::“`prototype`”}, final: **true**, enumerable: **false**, call: *Class_prototype*□

```
proc Class_prototype(this: OBJECT, phase: PHASE): OBJECT
    note this □ CLASS because this getter cannot be extracted from the Class class.
    prototype: OBJECTOPT □ this.prototype;
    if prototype = none then return undefined else return prototype end if
end proc;
```

ClassPrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE□localBindings: {
stdConstBinding(*public*::“`constructor`”, *Class*, *Class*),
stdFunction(*public*::“`toString`”, *Class_toString*, 0),
stdReserve(*public*::“`valueOf`”, *ObjectPrototype*),
stdConstBinding(*public*::“`length`”, *Number*, 1_{f64})},
archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**□

```
proc Class_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    note  This function does not check phase and therefore can be used in a constant expression.
    note  This function ignores any arguments passed to it in args.
    c: CLASS □  objectToClass(this);
    return "[class " ⊕ c.name ⊕ "]"
end proc;
```

17.19 Function

Function: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {*ivarFunctionLength*}, super: *Object*, prototype: *FunctionPrototype*, complete: **true**, name: "Function", typeofString: "function", dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

ivarFunctionLength: INSTANCEVARIABLE = **new** INSTANCEVARIABLE□multiname: {*public*::"length"}, final: **true**, enumerable: **false**, type: *Number*, defaultValue: **none**, immutable: **true**□

FunctionPrototype: SIMPLEINSTANCE = **new** SIMPLEINSTANCE□localBindings: {}, archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**□

17.19.1 PrototypeFunction

PrototypeFunction: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {**new** INSTANCEVARIABLE□multiname: {*public*::"prototype"}, final: **true**, enumerable: **false**, type: *Object*, defaultValue: **undefined**, immutable: **false**□}, super: *Function*, prototype: *FunctionPrototype*, complete: **true**, name: "Function", typeofString: "function", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

17.20 Package

Package: CLASS = **new** CLASS□localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *ObjectPrototype*, complete: **true**, name: "Package", typeofString: "object", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, hasProperty: *ordinaryHasProperty*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, coerce: *ordinaryCoerce*□

17.21 Error

```
Error: CLASS = new CLASS□localBindings: {}, instanceProperties: {
        new INSTANCEVARIABLE□multiname: {public::“name”}, final: false, enumerable: true, type: String,
        defaultValue: null, immutable: false□
        new INSTANCEVARIABLE□multiname: {public::“message”}, final: false, enumerable: true, type: String,
        defaultValue: null, immutable: false□},
        super: Object, prototype: ErrorPrototype, complete: true, name: “Error”, typeofString: “object”,
        dynamic: true, final: false, defaultValue: null, defaultHint: hintNumber, hasProperty: ordinaryHasProperty,
        bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
        bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
        enumerate: ordinaryEnumerate, call: callError, construct: ordinaryConstruct, init: initError, is: ordinaryIs,
        coerce: ordinaryCoerce□

proc callError(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT □ defaultArg(args, 0, undefined);
    if arg = null or is(arg, Error) then return arg
    else return construct(c, args, phase)
    end if
end proc;

proc initError(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    name: STRING □ NULL □ dotRead(ErrorPrototype, {public::“name”}, phase);
    Evaluate dotWrite(this, {public::“name”}, name, phase) and ignore its result;
    arg: OBJECT □ defaultArg(args, 0, undefined);
    message: STRING □ NULL;
    if arg = undefined then
        message □ dotRead(ErrorPrototype, {public::“message”}, phase)
    else message □ objectToString(arg, phase)
    end if;
    Evaluate dotWrite(this, {public::“message”}, message, phase) and ignore its result
end proc;

ErrorPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE□localBindings: {
        stdConstBinding(public::“constructor”, Class, Error),
        stdFunction(public::“toString”, Error_toString, 1),
        stdVarBinding(public::“name”, String, “Error”),
        stdVarBinding(public::“message”, String, an implementation-defined string)},
        archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
        env: none□

proc Error_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
    if phase = compile then
        throw a ConstantError exception — toString cannot be called on an Error from a constant expression
    end if;
    note  This function ignores any arguments passed to it in args.
    err: OBJECT □ coerceNonNull(this, Error);
    name: STRING □ NULL □ dotRead(err, {public::“name”}, phase);
    message: STRING □ NULL □ dotRead(err, {public::“message”}, phase);
    return an implementation-defined string derived from name, message, and optionally other properties of err
end proc;
```

```
proc systemError(e: CLASS, msg: STRING ∪ UNDEFINED): OBJECT
   return construct(e, [msg], run)
end proc;
```

17.21.1 Error Subclasses

```
proc makeBuiltInErrorSubclass(name: STRING): CLASS
   proc call(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
      if |args| > 1 then
         throw an ArgumentError exception — at most one argument can be supplied
      end if;
      arg: OBJECT ← defaultArg(args, 0, undefined);
      if arg = null or is(arg, Error) then return coerce(arg, c)
      else return construct(c, args, phase)
      end if
   end proc;
   c: CLASS ← new CLASS⟨⟨localBindings: {}, instanceProperties: {}, super: Error, complete: false, name: name,
         typeofString: “object”, dynamic: true, final: false, defaultValue: null, defaultHint: hintNumber,
         hasProperty: Error.hasProperty, bracketRead: Error.bracketRead, bracketWrite: Error.bracketWrite,
         bracketDelete: Error.bracketDelete, read: Error.read, write: Error.write, delete: Error.delete,
         enumerate: Error.enumerate, call: call, construct: ordinaryConstruct, init: none, is: ordinaryIs,
         coerce: ordinaryCoerce⟩⟩
   prototype: SIMPLEINSTANCE ← new SIMPLEINSTANCE⟨⟨localBindings: {
         stdConstBinding(public::“constructor”, Class, c),
         stdVarBinding(public::“name”, String, name),
         stdVarBinding(public::“message”, String, an implementation-defined string)},
         archetype: ErrorPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
         env: none⟩⟩
   proc init(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
      if |args| > 1 then
         throw an ArgumentError exception — at most one argument can be supplied
      end if;
      name2: STRING ∪ NULL ← dotRead(prototype, {public::“name”}, phase);
      Evaluate dotWrite(this, {public::“name”}, name2, phase) and ignore its result;
      arg: OBJECT ← defaultArg(args, 0, undefined);
      message: STRING ∪ NULL;
      if arg = undefined then message ← dotRead(prototype, {public::“message”}, phase)
      else message ← objectToString(arg, phase)
      end if;
      Evaluate dotWrite(this, {public::“message”}, message, phase) and ignore its result
   end proc;
   c.prototype ← prototype;
   c.init ← init;
   c.complete ← true;
   return c
end proc;
```

ArgumentError: CLASS = makeBuiltInErrorSubclass(“ArgumentError”);

AttributeError: CLASS = makeBuiltInErrorSubclass(“AttributeError”);

ConstantError: CLASS = makeBuiltInErrorSubclass(“ConstantError”);

DefinitionError: CLASS = makeBuiltInErrorSubclass(“DefinitionError”);

EvalError: CLASS = makeBuiltInErrorSubclass(“EvalError”);

RangeError: CLASS = makeBuiltInErrorSubclass(“RangeError”);

ReferenceError: CLASS = *makeBuiltInErrorSubclass*("ReferenceError");

SyntaxError: CLASS = *makeBuiltInErrorSubclass*("SyntaxError");

TypeError: CLASS = *makeBuiltInErrorSubclass*("TypeError");

UninitializedError: CLASS = *makeBuiltInErrorSubclass*("UninitializedError");

URIError: CLASS = *makeBuiltInErrorSubclass*("URIError");

A Index

A.1 Nonterminals

A.2 Tags

A.3 Semantic Domains

A.4 Globals