

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	3
2 Conformance	3
3 Normative References.....	3
4 Overview	3
5 Notational Conventions	3
5.1 Text.....	3
5.2 Semantic Domains	3
5.3 Tags	4
5.4 Booleans.....	4
5.5 Sets	4
5.6 Real Numbers.....	6
5.6.1 Bitwise Integer Operators.....	6
5.7 Characters.....	7
5.8 Lists	7
5.9 Strings.....	8
5.10 Tuples	9
5.11 Records.....	9
5.12 ECMAScript Numeric Types	10
5.12.1 Signed Long Integers.....	10
5.12.2 Unsigned Long Integers	10
5.12.3 Single-Precision Floating-Point Numbers	10
5.12.3.1 Shorthand Notation	11
5.12.3.2 Conversion.....	11
5.12.3.3 Arithmetic	12
5.12.4 Double-Precision Floating-Point Numbers	12
5.12.4.1 Shorthand Notation	13
5.12.4.2 Conversion.....	13
5.12.4.3 Arithmetic	13
5.13 Procedures	15
5.13.1 Operations	16
5.13.2 Semantic Domains of Procedures	16
5.13.3 Steps	16
5.13.4 Nested Procedures	18
5.14 Grammars	18
5.14.1 Grammar Notation	18
5.14.2 Lookahead Constraints	19
5.14.3 Line Break Constraints	19
5.14.4 Parameterised Rules	19
5.14.5 Special Lexical Rules	20
5.15 Semantic Actions	20
5.15.1 Example	21
5.15.2 Abbreviated Actions	22
5.15.3 Action Notation Summary	23
5.16 Other Semantic Definitions	24
6 Source Text.....	25
6.1 Unicode Format-Control Characters	25
7 Lexical Grammar	25
7.1 Input Elements	27
7.2 White space	28
7.3 Line Breaks	28
7.4 Comments.....	28
7.5 Keywords and Identifiers.....	29
7.6 Punctuators	32
7.7 Numeric literals	32
7.8 String literals	34
7.9 Regular expression literals.....	37
8 Program Structure.....	38
8.1 Packages	38
8.2 Scopes	38
9 Data Model	38
9.1 Objects	39
9.1.1 Undefined	39
9.1.2 Null	39
9.1.3 Booleans	40
9.1.4 Numbers	40
9.1.5 Strings	40
9.1.6 Namespaces	40
9.1.6.1 Qualified Names	40
9.1.7 Compound attributes	40
9.1.8 Classes	41
9.1.9 Method Closures	42
9.1.10 Prototype Instances	42
9.1.11 Class Instances	42
9.1.11.1 Open Instances	44
9.1.11.2 Slots	44
9.1.12 Packages	45
9.1.13 Global Objects	45
9.2 Objects with Limits	45
9.3 References	45
9.4 Function Support	46
9.5 Argument Lists	47
9.6 Modes of expression evaluation	47
9.7 Contexts	47
9.8 Labels	47
9.9 Environments	48
9.9.1 Frames	48
9.9.1.1 System Frame	48
9.9.1.2 Function Parameter Frames	48
9.9.1.3 Block Frames	49
9.9.2 Static Bindings	49
9.9.3 Instance Bindings	50
10 Data Operations	51
10.1 Numeric Utilities	51
10.2 Object Utilities	53
10.2.1 <i>objectType</i>	53
10.2.2 <i>hasType</i>	54
10.2.3 <i>toBoolean</i>	54
10.2.4 <i>toGeneralNumber</i>	55
10.2.5 <i>toString</i>	55
10.2.6 <i>toPrimitive</i>	57

10.2.7 Attributes.....	57	14.3 Import Directive.....	136
10.3 References.....	58	14.4 Pragma.....	136
10.4 Slots.....	60	15 Definitions.....	138
10.5 Environments.....	60	15.1 Export Definition	138
10.5.1 Access Utilities	60	15.2 Variable Definition	138
10.5.2 Adding Static Definitions.....	62	15.3 Simple Variable Definition.....	144
10.5.3 Adding Instance Definitions	63	15.4 Function Definition.....	145
10.5.4 Instantiation	65	15.5 Class Definition.....	149
10.5.5 Environmental Lookup.....	66	15.6 Namespace Definition.....	151
10.5.6 Property Lookup.....	67	15.7 Package Definition.....	151
10.5.7 Reading a Property	70	16 Programs.....	151
10.5.8 Writing a Property	74	17 Predefined Identifiers.....	152
10.5.9 Deleting a Property.....	77	18 Built-in Classes.....	152
10.6 Invocation.....	78	18.1 Object	154
11 Evaluation.....	78	18.2 Never	154
11.1 Phases of Evaluation.....	78	18.3 Void	154
11.2 Constant Expressions.....	78	18.4 Null	154
12 Expressions.....	78	18.5 Boolean.....	154
12.1 Identifiers	78	18.6 Integer.....	154
12.2 Qualified Identifiers.....	79	18.7 Number.....	154
12.3 Primary Expressions	81	18.7.1 ToNumber Grammar	154
12.4 Function Expressions.....	83	18.8 Character	154
12.5 Object Literals.....	83	18.9 String	154
12.6 Array Literals.....	85	18.10 Function.....	154
12.7 Super Expressions.....	86	18.11 Array	154
12.8 Postfix Expressions.....	87	18.12 Type	154
12.9 Member Operators	92	18.13 Math.....	154
12.10 Unary Operators.....	94	18.14 Date.....	154
12.11 Multiplicative Operators.....	97	18.15 RegExp	154
12.12 Additive Operators.....	99	18.15.1 Regular Expression Grammar	154
12.13 Bitwise Shift Operators	100	18.16 Error	154
12.14 Relational Operators	102	18.17 Attribute	154
12.15 Equality Operators	104	19 Built-in Functions.....	154
12.16 Binary Bitwise Operators	106	20 Built-in Attributes	154
12.17 Binary Logical Operators	109	21 Built-in Namespaces	155
12.18 Conditional Operator	110	22 Errors	155
12.19 Assignment Operators	112	23 Optional Packages	155
12.20 Comma Expressions	114	23.1 Machine Types	155
12.21 Type Expressions.....	115	23.2 Internationalisation	155
13 Statements	116	A Index	155
13.1 Empty Statement.....	119	A.1 Nonterminals	155
13.2 Expression Statement.....	119	A.2 Tags.....	156
13.3 Super Statement	120	A.3 Semantic Domains	156
13.4 Block Statement.....	120	A.4 Globals.....	157
13.5 Labeled Statements.....	121		
13.6 If Statement.....	122		
13.7 Switch Statement	123		
13.8 Do-While Statement	123		
13.9 While Statement	124		
13.10 For Statements	125		
13.11 With Statement	126		
13.12 Continue and Break Statements	126		
13.13 Return Statement.....	127		
13.14 Throw Statement.....	128		
13.15 Try Statement.....	128		
14 Directives.....	130		
14.1 Attributes.....	134		
14.2 Use Directive	135		

1 Scope

This Standard defines the ECMAScript Edition 4 scripting language.

2 Conformance

3 Normative References

4 Overview

5 Notational Conventions

This specification uses the notation below to represent algorithms and concepts. These concepts are used as notation only and are not necessarily represented or visible in the ECMAScript language.

5.1 Text

Throughout this document, the phrase *code point* and the word *character* is used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of Unicode text in the UTF-16 transformation format. The phrase *Unicode character* is used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code point). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual Unicode characters, even though a user might think of the whole sequence as a single character.

When denoted in this specification, characters with values between 20 and 7E hexadecimal inclusive are in a *fixed width font*. Other characters are denoted by enclosing their four-digit hexadecimal Unicode value between « and »». For example, the non-breakable space character would be denoted in this document as «u00A0». A few of the common control characters are represented by name:

Abbreviation	Unicode Value
«NUL»	«u0000»
«BS»	«u0008»
«TAB»	«u0009»
«LF»	«u000A»
«VT»	«u000B»
«FF»	«u000C»
«CR»	«u000D»
«SP»	«u0020»

A space character is denoted in this document either by a blank space where it's obvious from the context or by «SP» where the space might be confused with some other notation.

5.2 Semantic Domains

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

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

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

not a **true** if a is **false**; **false** if a is **true**

a **and** b If a is **false**, returns **false** without computing b ; if a is **true**, returns the value of b

a **or** b If a is **false**, returns the value of b ; if a is **true**, returns **true** without computing b

a **xor** b **true** if a is **true** and b is **false** or a is **false** and b is **true**; **false** otherwise. a **xor** b is equivalent to $a \neq b$

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

5.5 Sets

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

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

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

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

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

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

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

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

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

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

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,

$$\{x \mid x \in \text{INTEGER} \text{ such that } x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}$$

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

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

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

$x \in A$ **false** if x is an element of A and **true** if not

$|A|$ The number of elements in A (only used on finite sets)

$\min A$ The value m that satisfies both $m \in A$ and for all elements $x \in A, x \geq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)

$\max A$ The value m that satisfies both $m \in A$ and for all elements $x \in A, x \leq m$ (only used on nonempty, finite sets whose elements have a well-defined order relation)

$A \cap B$ The intersection of A and B (the set or semantic domain of all values that are present both in A and in B)

$A \cup B$ The union of A and B (the set or semantic domain of all values that are present in at least one of A or B)

$A - B$ The difference of A and B (the set or semantic domain of all values that are present in A but not B)

$A = B$ **true** if A and B are equal and **false** otherwise. A and B are equal if every element of A is also in B and every element of B is also in A .

$A \neq B$ **false** if A and B are equal and **true** otherwise

$A \subset B$ **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 $A \subset B$ and $A \neq B$.

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

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

then:

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

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

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

some $x \in A$ **satisfies** $\text{predicate}(x)$

returns **true** if there exists at least one element x in set A such that $\text{predicate}(x)$ 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 $\text{predicate}(x)$ computes to **true**; if there is more than one such element x , then one of them is chosen arbitrarily. For example,

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

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

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

The quantifier

$\text{every } x \in A \text{ satisfies } \text{predicate}(x)$

returns **true** if there exists no element x in set A such that $\text{predicate}(x)$ 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,

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

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 $\{-3, -2, -1, 0, 1, 2, 3, \dots\}$. 3.0 , 3 , $0xFF$, and -10^{100} are all integers.

RATIONAL is the semantic domain of all rational numbers. Every integer is also a rational number: **INTEGER** ⊑ **RATIONAL**. 3 , $1/3$, 7.5 , $-12/7$, and 2^{-5} are examples of rational numbers.

REAL is the semantic domain of all real numbers. Every rational number is also a real number: **RATIONAL** ⊑ **REAL**. π is an example of a real number slightly larger than 3.14 .

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

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

Real numbers can be compared using $=$, \neq , $<$, \leq , $>$, and \geq . 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:

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

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

$$a_i = \lfloor x / 2^i \rfloor \bmod 2$$

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, $\text{bitwiseAnd}(6, -6) = 2$.

$\text{bitwiseAnd}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER}$	<u>The Return the</u> bitwise AND of x and y
$\text{bitwiseOr}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER}$	<u>Return t</u> he bitwise OR of x and y
$\text{bitwiseXor}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER}$	<u>Return t</u> he bitwise XOR of x and y
$\text{bitwiseShift}(x: \text{INTEGER}, count: \text{INTEGER}): \text{INTEGER}$	<u>Shift</u> <u>Return x shifted</u> to the left by $count$ bits. If $count$ is negative, <u>shift return x shifted</u> to the right by $-count$ bits. Bits shifted out of the right end are lost; bit shifted in at the right end are zero. $\text{bitwiseShift}(x, count)$ is exactly equivalent to $\lfloor x \rfloor 2^{count}$

The Return the bitwise AND of x and y

Return the bitwise OR of x and y

Return the bitwise XOR of x and y

Shift Return x shifted to the left by $count$ bits. If $count$ is negative, shift 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.
 $\text{bitwiseShift}(x, count)$ is exactly equivalent to $\lfloor x \rfloor 2^{count}$

5.7 Characters

Characters enclosed in single quotes ‘ and ’ represent single Unicode 16-bit code points. Examples of characters include ‘A’, ‘b’, ‘«LF»’, and ‘«uFFFF»’ (see also section 5.1). Unicode surrogates are considered to be pairs of characters for the purpose of this specification.

CHARACTER is the semantic domain of all 65536 characters {‘«u0000»’ ... ‘«uFFFF»’}.

Characters can be compared using $=$, \neq , $<$, \leq , $>$, and \geq . These operators compare code point values, so ‘A’ = ‘A’, ‘A’ $<$ ‘B’, and ‘A’ $<$ ‘a’ are all **true**.

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

$\text{characterToCode}(c: \text{CHARACTER}): \{0 \dots 65535\}$	<u>Return character</u> c 's Unicode code point as an integer
$\text{codeToCharacter}(i: \{0 \dots 65535\}): \text{CHARACTER}$	<u>Return the character</u> whose Unicode code point is i

5.8 Lists

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

$[element_0, element_1, \dots, element_{n-1}]$

For example, the following list contains four strings:

$["parsley", "sage", "rosemary", "thyme"]$

The empty list is written as $[]$.

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

A list can also be written using the list comprehension notation

$[\mathbf{f}(x) | \square x \square u]$

which denotes the list $[\mathbf{f}(u[0]), \mathbf{f}(u[1]), \dots, \mathbf{f}(u[|u|-1])]$ 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:

$[\mathbf{f}(x) | \square x \square u \text{ such that } \mathbf{predicate}(x)]$

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

$$\begin{aligned} [x^2 | \square x \sqsubseteq [-1, 1, 2, 3, 4, 2, 5]] &= [1, 1, 4, 9, 16, 4, 25] \\ [x+1 | \square x \sqsubseteq [-1, 1, 2, 3, 4, 5, 3, 10]] \text{ such that } x \bmod 2 = 1 &= [0, 2, 4, 6, 4] \end{aligned}$$

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

Notation	Precondition	Description
$ u $		The length n of the list
$u[i]$	$0 \leq i < u $	The i^{th} element e_i .
$u[i \dots j]$	$0 \leq i \leq j+1 \leq u $	The list slice $[e_i, e_{i+1}, \dots, e_j]$ consisting of all elements of u between the i^{th} and the j^{th} , inclusive. The result is the empty list [] if $j=i-1$.
$u[i \dots]$	$0 \leq i \leq u $	The list slice $[e_i, e_{i+1}, \dots, e_{n-1}]$ 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 \leq i < u $	The list $[e_0, \dots, e_{i-1}, x, e_{i+1}, \dots, e_{n-1}]$ with the i^{th} element replaced by the value x and the other elements unchanged
$u \oplus v$		The concatenated list $[e_0, e_1, \dots, e_{n-1}, f_0, f_1, \dots, f_{m-1}]$
$\text{repeat}(e, i)$	$i \geq 0$	The list $[e, e, \dots, e]$ 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.
$u \neq v$		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 $x \sqsubseteq u$ satisfies $\text{predicate}(x)$
every $x \sqsubseteq u$ satisfies $\text{predicate}(x)$

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 $\text{predicate}(x)$. For example,

some $x \sqsubseteq [3, 36, 19, 26]$ satisfies $x \bmod 10 = 6$

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

5.9 Strings

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

“Wonder«LF”

is equivalent to:

[‘W’, ‘o’, ‘n’, ‘d’, ‘e’, ‘r’, ‘«LF»’]

The empty string is usually written as “”.

In addition to the other list operations, $<$, \leq , $>$, and \geq are defined on strings. A string x is less than string y when y is not the empty string and either x is the empty string, the first character of x is less than the first character of y , or the first character of x is equal to the first character of y and the rest of string x is less than the rest of string y .

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

5.10 Tuples

A *tuple* is an immutable aggregate of values comprised of a name **NAME** and zero or more labelled fields.

The fields of each kind of tuple used in this specification are described in tables such as:

Field	Contents	Note
label ₁	T ₁	Informative note about this field
...
label _n	T _n	Informative note about this field

label₁ 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**₁: *v*₁, ..., **label**_k: *v*_k, other fields from *a*]

which represents a tuple with name **NAME** and values *v*₁ through *v*_k for fields labeled **label**₁ through **label**_k respectively and the values of correspondingly labeled fields from *a* for all other fields.

If *a* is the tuple **NAME** [**label**₁: *v*₁, ..., **label**_n: *v*_n] then

a.**label**_i
returns the *i*th field's value *v*_i.

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

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

5.11 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

label₁ 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 $\boxed{label_{i1}: v_{i1}, \dots, label_{ik}: v_{ik}}$, other fields from a

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

If a is a record with name **NAME** and address $\boxed{}$, then

$a.label_i$

returns the current value v of the i^{th} field at address $\boxed{}$. That field may be set to a new value w , which must be a member of the semantic domain T_i , using the assignment

$a.label_i \boxed{} w$

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

The equality operators $=$ and \neq 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** \sqcup **ULONG** \sqcup **FLOAT32** \sqcup **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** \sqcup **ULONG** \sqcup **FLOAT32** \sqcup **FLOAT64**

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	$\{-2^{63} \dots 2^{63} - 1\}$	The signed 64-bit integer

5.12.2 Unsigned Long Integers

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

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

5.12.3 Single-Precision Floating-Point Numbers

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

```
FLOAT32 = FINITEFLOAT32 ∪ {+∞f32, -∞f32, NaNf32};  
FINITEFLOAT32 = NONZEROFINITEFLOAT32 ∪ {+zerof32, -zerof32}
```

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
<code>value</code>	<code>NORMALISEDFLOAT32VALUES</code> ∪ <code>DENORMALISEDFLOAT32VALUES</code>	The value, represented as an exact rational number

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

```
NORMALISEDFLOAT32VALUES = {s|m|2e | |s| {-1, 1}, |m| {223 ... 224-1}, |e| {-149 ... 104}}
```

`m` is called the significand.

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

```
DENORMALISEDFLOAT32VALUES = {s|m|2-149 | |s| {-1, 1}, |m| {1 ... 223-1}}
```

`m` is called the significand.

The remaining `FLOAT32` values are the tags `+zerof32` (positive zero), `-zerof32` (negative zero), `+∞f32` (positive infinity), `-∞f32` (negative infinity), and `NaNf32` (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 `+zerof32 ≠ -zerof32` but `NaNf32 = NaNf32`. The ECMAScript `x == y` and `x === y` operators have different behavior for `FLOAT32` values, defined by `isEqual` and `isStrictEqual`.

5.12.3.1 Shorthand Notation

In this specification, when `x` is a real number or expression, the notation `xf32` indicates the result of `realToFloat32(x)`, which is the “closest” `FLOAT32` value as defined below. Thus, 3.4 is a `REAL` number, while `3.4f32` is a `FLOAT32` value (whose exact `value` is actually 3.400000095367431640625). The positive finite `FLOAT32` values range from 10^{-45}_{f32} to $(3.4028235 \times 10^{38})_{f32}$.

5.12.3.2 Conversion

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

```
proc realToFloat32(x: REAL): FLOAT32  
  s: RATIONAL {} ∪ NORMALISEDFLOAT32VALUES ∪ DENORMALISEDFLOAT32VALUES ∪ {-2128, 0, 2128};  
  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 -2128, 0, and 2128 are considered to have  
  even significands.  
  if a = 2128 then return +∞f32  
  elseif a = -2128 then return -∞f32  
  elseif a ≠ 0 then return NONZEROFINITEFLOAT32[value: a]  
  elseif x < 0 then return -zerof32  
  else return +zerof32  
  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 ∉ {+zerof32, -zerof32} then return 0 end if;  
  r: RATIONAL ∪ x.value;  
  if r > 0 then return ⌊r⌋ else return ⌈r⌉ end if  
end proc
```

5.12.3.3 Arithmetic

The following table defines ~~s procedures that perform common arithmetic on negation of FLOAT32 values using IEEE 754 rules~~. Note that $(expr)_{f32}$ is a shorthand for $realToFloat32(expr)$.

~~float32Abs(x: FLOAT32): FLOAT32~~

~~float32Negate(x: FLOAT32): FLOAT32~~

x	Result
$-\infty_{f32}$	$+\infty_{f32}$
negative finite	$(-x.value)_{f32}$
$-zero_{f32}$	$+zero_{f32}$
$+zero_{f32}$	$-zero_{f32}$
positive finite	$(-x.value)_{f32}$
$+\infty_{f32}$	$-\infty_{f32}$
NaN_{f32}	NaN_{f32}

~~float32Add(x: FLOAT32, y: FLOAT32): FLOAT32~~

~~The identity for floating point addition is $-zero$, not $+zero$.~~

~~float32Subtract(x: FLOAT32, y: FLOAT32): FLOAT32~~

~~float32Multiply(x: FLOAT32, y: FLOAT32): FLOAT32~~

~~float32Divide(x: FLOAT32, y: FLOAT32): FLOAT32~~

~~float32Remainder(x: FLOAT32, y: FLOAT32): FLOAT32~~

~~Note that $float32Remainder(float32Negate(x), y)$ always produces the same result as $float32Negate(float32Remainder(x, y))$. Also, $float32Remainder(x, float32Negate(y))$ always produces the same result as $float32Remainder(x, y)$.~~

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 $\sqcup \{+\infty_{f64}, -\infty_{f64}, \text{NaN}_{f64}\}$;

FINITEFLOAT64 = NONZEROFINITEFLOAT64 $\sqcup \{+zero_{f64}, -zero_{f64}\}$

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 \sqcup DENORMALISEDFLOAT64VALUES	The value, represented as an exact rational number

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

$\text{NORMALISEDFLOAT64VALUES} = \{s \square m \square 2^e \mid s \in \{-1, 1\}, m \in \{2^{52} \dots 2^{53}-1\}, e \in \{-1074 \dots 971\}\}$

m is called the significand.

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

$\text{DENORMALISEDFLOAT64VALUES} = \{s \square m \square 2^{-1074} \mid s \in \{-1, 1\}, m \in \{1 \dots 2^{52}-1\}\}$

m is called the significand.

The remaining FLOAT64 values are the tags $+zero_{f64}$ (positive zero), $-zero_{f64}$ (negative zero), $+\infty_{f64}$ (positive infinity), $-\infty_{f64}$ (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 `+zerof64` \neq `-zerof64` but `NaNf64` = `NaNf64`. The ECMAScript `x == y` and `x === y` operators have different behavior for `FLOAT64` values, defined by `isEqual` and `isStrictEqual`.

5.12.4.1 Shorthand Notation

In this specification, when `x` is a real number or expression, the notation `xf64` indicates the result of `realToFloat64(x)`, which is the “closest” `FLOAT64` value as defined below. Thus, 3.4 is a `REAL` number, while `3.4f64` is a `FLOAT64` value (whose exact `value` is actually `3.39999999999999911182158029987476766109466552734375`). 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 ⊔ {−21024, 0, 21024};
  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 -zerof64
  else return +zerof64
  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`

<i>x</i>	Result
<code>-∞_{f32}</code>	<code>-∞_{f64}</code>
<code>-zero_{f32}</code>	<code>-zero_{f64}</code>
<code>+zero_{f32}</code>	<code>+zero_{f64}</code>
<code>+∞_{f32}</code>	<code>+∞_{f64}</code>
<code>NaN_{f32}</code>	<code>NaN_{f64}</code>
Any <code>NONZEROFINITEFLOAT32</code> value	<code>NONZEROFINITEFLOAT64[value: x.value]</code>

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

```
proc truncateFiniteFloat64(x: FINITEFLOAT64): INTEGER
  if x ⊔ {+zerof64, -zerof64} then return 0 end if;
  r: RATIONAL ⊔ x.value;
  if r > 0 then return r else return r end if
end proc
```

5.12.4.3 Arithmetic

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

float64Abs(x: FLOAT64): FLOAT64

<i>x</i>	Result
$-\infty_{f64}$	$+\infty_{f64}$
negative finite	$(-x.value)_{f64}$
$-zero_{f64}$	$+zero_{f64}$
$+zero_{f64}$	$+zero_{f64}$
positive finite	<i>x</i>
$+\infty_{f64}$	$+\infty_{f64}$
NaN_{f64}	NaN_{f64}

float64Negate(x: FLOAT64): FLOAT64

<i>x</i>	Result
$-\infty_{f64}$	$+\infty_{f64}$
negative finite	$(-x.value)_{f64}$
$-zero_{f64}$	$+zero_{f64}$
$+zero_{f64}$	$-zero_{f64}$
positive finite	$(-x.value)_{f64}$
$+\infty_{f64}$	$-\infty_{f64}$
NaN_{f64}	NaN_{f64}

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

NOTE The identity for floating-point addition is **-zero_{f64}**, not **+zero_{f64}**.

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

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

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

<i>x</i>	<i>y</i>						
	$-\infty_{f64}$	negative finite	$-zero_{f64}$	$+zero_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}	NaN_{f64}	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}
negative finite	$+\infty_{f64}$	$(x.\text{value} \sqcup y.\text{value})_{f64}$	$+zero_{f64}$	$-zero_{f64}$	$(x.\text{value} \sqcup y.\text{value})_{f64}$	$-\infty_{f64}$	NaN_{f64}
$-zero_{f64}$	NaN_{f64}	$+zero_{f64}$	$+zero_{f64}$	$-zero_{f64}$	$-zero_{f64}$	NaN_{f64}	NaN_{f64}
$+zero_{f64}$	NaN_{f64}	$-zero_{f64}$	$-zero_{f64}$	$+zero_{f64}$	$+zero_{f64}$	NaN_{f64}	NaN_{f64}
positive finite	$-\infty_{f64}$	$(x.\text{value} \sqcup y.\text{value})_{f64}$	$-zero_{f64}$	$+zero_{f64}$	$(x.\text{value} \sqcup y.\text{value})_{f64}$	$+\infty_{f64}$	NaN_{f64}
$+\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}	NaN_{f64}	$+\infty_{f64}$	$+\infty_{f64}$	NaN_{f64}
NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}	NaN_{f64}

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

<i>x</i>	<i>y</i>						
	$-\infty_{f64}$	negative finite	$-zero_{f64}$	$+zero_{f64}$	positive finite	$+\infty_{f64}$	NaN_{f64}
$-\infty_{f64}$	NaN_{f64}	$+\infty_{f64}$	$+\infty_{f64}$	$-\infty_{f64}$	$-\infty_{f64}$	NaN_{f64}	NaN_{f64}
negative finite	$+zero_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$+zero_{f64}$	$-\infty_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$-zero_{f64}$	NaN_{f64}
$-zero_{f64}$	$+zero_{f64}$	$+zero_{f64}$	NaN_{f64}	NaN_{f64}	$-zero_{f64}$	$-zero_{f64}$	NaN_{f64}
$+zero_{f64}$	$-zero_{f64}$	$-zero_{f64}$	NaN_{f64}	NaN_{f64}	$+zero_{f64}$	$+zero_{f64}$	NaN_{f64}
positive finite	$-zero_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$-\infty_{f64}$	$+\infty_{f64}$	$(x.\text{value} / y.\text{value})_{f64}$	$+zero_{f64}$	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

<i>x</i>	<i>y</i>							
	$-\infty_{f64}, +\infty_{f64}$	positive or negative finite	$-zero_{f64}, +zero_{f64}$	NaN_{f64}				
$-\infty_{f64}$	NaN_{f64}	NaN_{f64}			NaN_{f64}	NaN_{f64}		
negative finite	x	$float64Negate(float64Remainder(float64Negate(x), y))$		NaN_{f64}				
$-zero_{f64}$	$-zero_{f64}$	$-zero_{f64}$			NaN_{f64}	NaN_{f64}		
$+zero_{f64}$	$+zero_{f64}$	$+zero_{f64}$			NaN_{f64}	NaN_{f64}		
positive finite	x	$(x.\text{value} - y.\text{value}) \sqcup (x.\text{value} / y.\text{value})_{f64}$			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 that $float64Remainder(float64Negate(x), y)$ always produces the same result as $float64Negate(float64Remainder(x, y))$. Also, $float64Remainder(x, float64Negate(y))$ always produces the same result as $float64Remainder(x, y)$.

5.13 Procedures

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

A procedure is denoted as:

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

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

`f` is the procedure's name, `param1` through `paramn` are the procedure's parameters, `T1` through `Tn` are the parameters' respective semantic domains, `T` is the semantic domain of the procedure's result, and `step1` through `stepm` 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 `v1` through `vn`, the procedure's steps are performed and the result, if any, returned to the caller.

A procedure's steps can refer to the parameters `param1` through `paramn`; each reference to a parameter `parami` evaluates to the corresponding argument value `vi`. 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(arg1, ..., argn)` syntax. `f` is computed first, followed by the argument expressions `arg1` through `argn`, 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 `T1` through `Tn` respectively and produce a result in semantic domain `T` is written as `T1 ⊔ T2 ⊔ ... ⊔ Tn ⊔ T`. If `n = 0`, this semantic domain is written as `() ⊔ T`. If the procedure does not produce a result, the semantic domain of procedures is written either as `T1 ⊔ T2 ⊔ ... ⊔ Tn ⊔ ()` or as `() ⊔ ()`.

5.13.3 Steps

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

nothing

A `nothing` step performs no operation.

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 expression1 then step; step; ...; step
elseif expression2 then step; step; ...; step
...
elseif expressionn 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 *steps* is performed. Otherwise, *expression*₂ is computed and tested, and so on. If no *expression* evaluates to **true**, the list of *steps* following the **else** is performed. The **else** clause may be omitted, in which case no action is taken when no *expression* evaluates to **true**.

```

case expression of
  T1 do step; step; ...; step;
  T2 do step; step; ...; step;
  ...
  Tn do step; step; ...; step
  else step; step; ...; step
end case

```

A **case** step computes *expression*, which will evaluate to a value *v*. If *v* \in T₁, then the first list of *steps* is performed. Otherwise, if *v* \in T₂, then the second list of *steps* is performed, and so on. If *v* is not a member of any T_{*i*}, the list of *steps* 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 *steps* 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  $\sqsubseteq$  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 *steps* 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 *steps*. If they complete normally (or if they **return** out of the current procedure), then the **try** step is done. If any of the *steps* 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 *steps* 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 *steps*.

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
  | SampleListPrefix
  | SampleListPrefix , ... Identifier
```

(*Identifier*: 12.1)

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] ListExpressionallowIn
```

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

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

5.14.4 Parameterised Rules

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

Metadefinitions such as

```
□ □ {normal, initial}
```

$\square \square \{allowIn, noIn\}$

introduce grammar arguments \square and \square . 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

```
AssignmentExpressionallowIn, noIn  $\square$ 
  ConditionalExpressionallowIn, noIn
  | LeftSideExpressionallowIn, noIn = AssignmentExpressionnormal, allowIn
  | LeftSideExpressionallowIn, noIn CompoundAssignment AssignmentExpressionnormal, allowIn
```

expands into the following four rules:

```
AssignmentExpressionnormal, allowIn  $\square$ 
  ConditionalExpressionnormal, allowIn
  | LeftSideExpressionnormal = AssignmentExpressionnormal, allowIn
  | LeftSideExpressionnormal CompoundAssignment AssignmentExpressionnormal, allowIn
```

```
AssignmentExpressionnormal, noIn  $\square$ 
  ConditionalExpressionnormal, noIn
  | LeftSideExpressionnormal = AssignmentExpressionnormal, noIn
  | LeftSideExpressionnormal CompoundAssignment AssignmentExpressionnormal, noIn
```

```
AssignmentExpressioninitial, allowIn  $\square$ 
  ConditionalExpressioninitial, allowIn
  | LeftSideExpressioninitial = AssignmentExpressionnormal, allowIn
  | LeftSideExpressioninitial CompoundAssignment AssignmentExpressionnormal, allowIn
```

```
AssignmentExpressioninitial, noIn  $\square$ 
  ConditionalExpressioninitial, noIn
  | LeftSideExpressioninitial = AssignmentExpressionnormal, noIn
  | LeftSideExpressioninitial CompoundAssignment AssignmentExpressionnormal, noIn
```

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

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

5.14.5 Special Lexical Rules

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

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:

$NonAsteriskOrSlash \square \text{ UnicodeCharacter except } * \mid /$

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 \sqsubseteq 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Digits \sqsubseteq
 \sqcup *Digit*
 \sqcup *Digits Digit*

Numeral \sqsubseteq
 \sqcup *Digits*
 \sqcup *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] \sqcup Digit = *Value[Digit]*;

DecimalValue[Digits₀] \sqcup Digits₁ Digit = 10 \lceil *DecimalValue[Digits₁] + Value[Digit]*;

proc *BaseValue[Digits] (base: INTEGER)*: INTEGER

[Digits \sqcup Digit] do

d: INTEGER \sqsubseteq Value[Digit];

if *d < base* **then return** *d* **else throw syntaxError** **end if**;

[Digits₀] \sqcup Digits₁ Digit] do

d: INTEGER \sqsubseteq Value[Digit];

if *d < base* **then return** *base* \lceil *BaseValue[Digits₁](base) + d*

else throw syntaxError

end if

end proc;

Value[Numeral]: INTEGER;

Value[Numeral] \sqcup Digits = *DecimalValue[Digits]*;

Value[Numeral] \sqcup Digits₁ # Digits₂

begin

base: INTEGER \sqsubseteq DecimalValue[Digits₂];

if *base ≥ 2 and base ≤ 10* **then return** *BaseValue[Digits₁](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:

<u>Input</u>	<u>Semantic Result</u>
<u>37</u>	<u>37</u>
<u>33#4</u>	<u>15</u>
<u>30#2</u>	<u>throw syntaxError</u>

5.15.2 Abbreviated Actions

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

Given the sample grammar rule

Expression □

Subexpression

| Expression * Subexpression
| Subexpression + Subexpression
| this

the notation

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

is an abbreviation for the following:

```
proc Validate[Expression] (cxt: CONTEXT, env: ENVIRONMENT)
  [Expression □ Subexpression] do Validate[Subexpression](cxt, env);
  [Expression0 □ Expression1 * Subexpression] do
    Validate[Expression1](cxt, env);
    Validate[Subexpression](cxt, env);
  [Expression □ Subexpression1 + Subexpression2] do
    Validate[Subexpression1](cxt, env);
    Validate[Subexpression2](cxt, env);
  [Expression □ this] do nothing
end proc;
```

Note that:

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

5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

Action[nonterminal]: T:

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

Action[nonterminal] □ expansion = expression;

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

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

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

Action[nonterminal] □ expansion]

```
begin
  step1:
  step2:
  ...
  stepn
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 *step₁* through *step_n*. A **return** step produces the value of the action.

```
proc Action[nonterminal □ expansion] (param1: T1, ..., paramn: Tn): T
  step1;
  step2;
  ...
  stepn
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 *step₁* through *step_m*.

```
proc Action[nonterminal] (param1: T1, ..., paramn: Tn): T
  [nonterminal □ expansion1] do
    step;
    ...
    step;
  [nonterminal □ expansion2] do
    step;
    ...
    step;
  ...
  [nonterminal □ expansionn] do
    step;
    ...
    step
end proc;
```

This notation is used only when Action returns a procedure when applied to nonterminal *nonterminal* with several expansions *expansion₁* 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] (param₁: T₁, ..., 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(param1: T1, ..., paramn: Tn): T
  step1;
  step2;
  ...
  stepm
end proc;
```

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

6 Source Text

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

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

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

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

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

6.1 Unicode Format-Control Characters

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

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

7 Lexical Grammar

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

A *token* is one of the following:

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

- One of the non-reserved words `exclude`, `get`, `include`, `named`, `set`.
- A punctuator token, which is one of `!`, `!=`, `!==`, `%`, `%=`, `&`, `&&`, `&=`, `(`, `)`, `*`, `*=`, `+`, `++`, `+=`, `,`, `-`, `--`, `-=`, `..`, `....`, `/`, `/=`, `:`, `::`, `:`, `<`, `<<`, `<=`, `=`, `==`, `====`, `>`, `>=`, `>>`, `>>=`, `>>>`, `=`, `?`, `[`, `]`, `^`, `^=`, `^^`, `^=`, `{`, `|`, `|=`, `||`, `||=`, `}`, `~`.
- An **Identifier** token, which carries a **STRING** that is the identifier's name.
- A **Number** token, which carries a **GENERALNUMBER** that is the number's value.
- A **NegatedMinLong** token, which carries no value. This token is the result of evaluating `9223372036854775808L`.
- A **String** token, which carries a **STRING** that is the string's value.
- A **RegularExpression** token, which carries two **STRINGS** — the regular expression's body and its flags.

A **LineBreak**, although not considered to be a token, also becomes part of the stream of input elements and guides the process of automatic semicolon insertion (section *****). **EndOfInput** signals the end of the source text.

NOTE The lexical grammar discards simple white space and single-line comments. They do not appear in the stream of input elements for the syntactic grammar. Comments spanning several lines become **LineBreaks**.

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

$$\text{InputElement} = \{\text{LineBreak}, \text{EndOfInput}\} \sqcup \text{TOKEN}$$

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

```

NextInputElementre □ WhiteSpace InputElementre                                (WhiteSpace: 7.2)
NextInputElementdiv □ WhiteSpace InputElementdiv

NextInputElementnum □ [lookahead {ContinuingIdentifierCharacter, \}] WhiteSpace InputElementdiv

InputElementre □
  | 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

InputElementdiv □
  | 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[NextInputElementre] = Lex[InputElementre];
  Lex[NextInputElementdiv] = Lex[InputElementdiv];
  Lex[NextInputElementnum] = Lex[InputElementdiv];
    = Lex[InputElementdiv];

Lex[InputElement]: INPUTELEMENT;
  Lex[InputElement] □ LineBreaks = LineBreak;
  Lex[InputElement] □ IdentifierOrKeyword = Lex[IdentifierOrKeyword];
  Lex[InputElement] □ Punctuator = Lex[Punctuator];
  Lex[InputElementdiv] □ DivisionPunctuator = Lex[DivisionPunctuator];
  Lex[InputElement] □ NumericLiteral = Lex[NumericLiteral];
  Lex[InputElement] □ StringLiteral = Lex[StringLiteral];
  Lex[InputElementre] □ 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

 | *LineBreaksWhiteSpace 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 \sqsubseteq $/ * \text{ MultiLineBlockCommentCharacters } \text{ BlockCommentCharacters } * /$

MultiLineBlockCommentCharacters \sqsubseteq

BlockCommentCharacters LineTerminator

(LineTerminator: 7.3)

$| \text{ MultiLineBlockCommentCharacters BlockCommentCharacters LineTerminator}$

UnicodeCharacter \sqsubseteq Any Unicode character

NonTerminator \sqsubseteq *UnicodeCharacter except LineTerminator*

NonTerminatorOrSlash \sqsubseteq *NonTerminator except /*

NonTerminatorOrAsteriskOrSlash \sqsubseteq *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 \sqsubseteq *IdentifierName*

Semantics

Lex[IdentifierOrKeyword \sqsubseteq IdentifierName]: INPUTELEMENT

begin

id: STRING \sqsubseteq LexName[IdentifierName];
if id \sqsubseteq {"abstract", "as", "break", "case", "catch", "class", "const", "continue", "debugger", "default", "delete", "do", "else", "enum", "exclude", "export", "extends", "false", "final", "finally", "for", "function", "get", "goto", "if", "implements", "import", "in", "include", "instanceof", "interface", "is", "named", "namespace", "native", "new", "null", "package", "private", "protected", "public", "return", "set", "static", "super", "switch", "synchronized", "this", "throw", "throws", "transient", "true", "try", "typeof", "use", "var", "volatile", "while", "with"}
and IdentifierName contains no escape sequences (i.e. expansions of the NullEscape or HexEscape nonterminals)
then return the keyword token id
else return an Identifier token with the name id
end if
end:

Lex[IdentifierOrKeyword \sqsubseteq IdentifierName]

Let id be the string LexString[IdentifierName].

If IdentifierName contains no escape sequences (i.e. expansions of the NullEscape or HexEscape nonterminals) and exactly matches one of the keywords abstract, as, break, case, catch, class, const, continue, debugger, default, delete, do, else, enum, exclude, export, extends, false, final, finally, for, function, get, goto, if, implements, import, in, include, instanceof, interface, is, namespace, named, native, new, null, package, private, protected, public, return, set, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, use, var, void, volatile, while, with, then return a keyword token with string contents id.

Return an identifier token with string contents id.

NOTE Even though the lexical grammar treats `exclude`, `get`, `include`, `named`, and `set` as keywords, the syntactic grammar contains productions that permit them to be used as identifier names. The other keywords are reserved and may not be used as

identifier names. However, an *IdentifierName* can never be a keyword if it contains any escape characters, so, for example, one can use `new` as the name of an identifier by including an escape sequence in it; `_new` is one possibility, and `n\x65w` is another.

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), `Li` (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), `Li` (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

(*HexEscape*: 7.8)

Semantics

LexName[IdentifierName]: STRING;

LexName[IdentifierName] □ InitialIdentifierCharacterOrEscape = *[LexChar[InitialIdentifierCharacterOrEscape]]*;

LexName[IdentifierName] □ NullEscapes InitialIdentifierCharacterOrEscape

= *[LexChar[InitialIdentifierCharacterOrEscape]]*;

LexName[IdentifierName] □ IdentifierName₀ ContinuingIdentifierCharacterOrEscape

= *[LexName[IdentifierName₁] ⊕ [LexChar[ContinuingIdentifierCharacterOrEscape]]]*;

LexName[IdentifierName] □ IdentifierName₀ NullEscape = *LexName[IdentifierName₁]*;

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:

LexString[IdentifierName □ -InitialIdentifierCharacterOrEscape]
LexString[IdentifierName □ -NullEscapes InitialIdentifierCharacterOrEscape]
 Return a one character string with the character LexChar[InitialIdentifierCharacterOrEscape].

LexString[IdentifierName □ IdentifierName₁ ContinuingIdentifierCharacterOrEscape]
 Return a string consisting of the string LexString[IdentifierName₁] concatenated with the character LexChar[ContinuingIdentifierCharacterOrEscape].

LexString[IdentifierName □ IdentifierName₁ NullEscape]
 Return the string LexString[IdentifierName₁].

LexChar[InitialIdentifierCharacterOrEscape □ -InitialIdentifierCharacter]
 Return the character InitialIdentifierCharacter.

LexChar[InitialIdentifierCharacterOrEscape □ \ HexEscape]
 Let ch be the character LexChar[HexEscape].
 If ch is in the set of characters accepted by the nonterminal InitialIdentifierCharacter, then return ch.
 Signal a syntax error.

LexChar[ContinuingIdentifierCharacterOrEscape]: 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:

LexChar[ContinuingIdentifierCharacterOrEscape □ -ContinuingIdentifierCharacter]
 Return the character ContinuingIdentifierCharacter.

LexChar[ContinuingIdentifierCharacterOrEscape □ \ HexEscape]
 Let ch be the character LexChar[HexEscape].
 If ch is in the set of characters accepted by the nonterminal ContinuingIdentifierCharacter, then return ch.
 Signal a syntax error.

The characters in the specified categories in version 2-13.0 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 \sqcup

!	! =	! ==	%	% =	&	& &
& & =	& =	()	* =	* =	+
+ +	+ =	,	-	--	- =	.
. . .	: =	: :	;	<	<<	<< =
< =	=	==	===	>	> =	>>
> > =	> > >	> > > =	?	[]	^
^ =	^ ^	^ ^ =	{		=	
=	} =	~				

DivisionPunctuator \sqcup

/ [lookahead { /, *}]
/=

Semantics

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

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

7.7 Numeric literals

Syntax

NumericLiteral \sqcup

DecimalLiteral
HexIntegerLiteral
DecimalLiteral LetterF
IntegerLiteral LetterL
IntegerLiteral LetterU LetterL

IntegerLiteral \sqcup

DecimalIntegerLiteral
HexIntegerLiteral

LetterF \sqcup F | f

LetterL \sqcup L | l

LetterU \sqcup U | u

DecimalLiteral \sqcup

Mantissa
Mantissa LetterE SignedInteger

LetterE \sqcup E | e

Mantissa \sqcup

DecimalIntegerLiteral
DecimalIntegerLiteral .
DecimalIntegerLiteral . Fraction
. Fraction

```

DecimalIntegerLiteral ◻
  ◻ 0
  | NonZeroDecimalDigits

NonZeroDecimalDigits ◻
  ◻ NonZeroDigit
  | NonZeroDecimalDigits ASCIIigit

Fraction ◻ DecimalDigits

SignedInteger ◻
  ◻ DecimalDigits
  | + DecimalDigits
  | - DecimalDigits

DecimalDigits ◻
  ◻ ASCIigit
  | DecimalDigits ASCIIigit

HexIntegerLiteral ◻
  ◻ 0 LetterX HexDigit
  | HexIntegerLiteral HexDigit

LetterX ◻ x | X

ASCIigit ◻ 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 ≤ 263 – 1 then return a Number token with the value LONG[value: i]
  elseif i = 263 then return NegatedMinLong
  else throw rangeError
  end if
end;
Lex[NumericLiteral ◻ IntegerLiteral LetterU LetterL]
begin
  i: INTEGER ◻ LexNumber[IntegerLiteral];
  if i ≤ 264 – 1 then return a Number token with the value ULONG[value: i] else throw rangeError end if
end;

LexNumber[IntegerLiteral]: INTEGER;
LexNumber[IntegerLiteral ◻ DecimalIntegerLiteral] = LexNumber[DecimalIntegerLiteral];
LexNumber[IntegerLiteral ◻ HexIntegerLiteral] = LexNumber[HexIntegerLiteral];

```

NOTE Note that all digits of hexadecimal literals are significant.

LexNumber[DecimalLiteral]: RATIONAL;

LexNumber[DecimalLiteral □ Mantissa] = LexNumber[Mantissa];

LexNumber[DecimalLiteral □ Mantissa LetterE SignedInteger] = LexNumber[Mantissa]□10^{LexNumber[SignedInteger]};

LexNumber[Mantissa]: RATIONAL;

LexNumber[Mantissa □ DecimalIntegerLiteral] = LexNumber[DecimalIntegerLiteral];

LexNumber[Mantissa □ DecimalIntegerLiteral .] = LexNumber[DecimalIntegerLiteral];

LexNumber[Mantissa □ DecimalIntegerLiteral . Fraction]

= LexNumber[DecimalIntegerLiteral] + LexNumber[Fraction];

LexNumber[Mantissa □ . Fraction] = LexNumber[Fraction];

LexNumber[DecimalIntegerLiteral]: INTEGER;

LexNumber[DecimalIntegerLiteral □ 0] = 0;

LexNumber[DecimalIntegerLiteral □ NonZeroDecimalDigits] = LexNumber[NonZeroDecimalDigits];

LexNumber[NonZeroDecimalDigits]: INTEGER;

LexNumber[NonZeroDecimalDigits □ NonZeroDigit] = DecimalValue[NonZeroDigit];

LexNumber[NonZeroDecimalDigits₀ □ NonZeroDecimalDigits₁ ASCIIDigit]

= 10□LexNumber[NonZeroDecimalDigits₁] + DecimalValue[ASCIIDigit];

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

LexNumber[SignedInteger]: INTEGER;

LexNumber[SignedInteger □ DecimalDigits] = LexNumber[DecimalDigits];

LexNumber[SignedInteger □ + DecimalDigits] = LexNumber[DecimalDigits];

LexNumber[SignedInteger □ - DecimalDigits] = -LexNumber[DecimalDigits];

LexNumber[DecimalDigits]: INTEGER;

LexNumber[DecimalDigits □ ASCIIDigit] = DecimalValue[ASCIIDigit];

LexNumber[DecimalDigits₀ □ DecimalDigits₁ ASCIIDigit]

= 10□LexNumber[DecimalDigits₁] + DecimalValue[ASCIIDigit];

NDigits[DecimalDigits]: INTEGER;

NDigits[DecimalDigits □ ASCIIDigit] = 1;

NDigits[DecimalDigits₀ □ DecimalDigits₁ ASCIIDigit] = NDigits[DecimalDigits₁] + 1;

LexNumber[HexIntegerLiteral]: INTEGER;

LexNumber[HexIntegerLiteral □ 0 LetterX HexDigit] = HexValue[HexDigit];

LexNumber[HexIntegerLiteral₀ □ HexIntegerLiteral₁ HexDigit]

= 16□LexNumber[HexIntegerLiteral₁] + 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 \square can be either **single** or **double**.

<i>StringLiteral</i> \square		
' ' <i>StringChars</i> ^{single} '		
' ' " <i>StringChars</i> ^{double} "		
<i>StringChars</i> [□]		
«empty»		
' ' <i>StringChars</i> [□] <i>StringChar</i> [□]		(NullEscape: 7.5)
' ' <i>StringChars</i> [□] <i>NullEscape</i>		
<i>StringChar</i> [□]		
<i>LiteralStringChar</i> [□]		
' ' \ <i>StringEscape</i>		
<i>LiteralStringChar</i> ^{single} \square <i>UnicodeCharacter</i> except ' \ <i>LineTerminator</i>		(UnicodeCharacter: 7.3)
<i>LiteralStringChar</i> ^{double} \square <i>UnicodeCharacter</i> except " \ <i>LineTerminator</i>		(LineTerminator: 7.3)
<i>StringEscape</i> \square		
<i>ControlEscape</i>		
' ' <i>ZeroEscape</i>		
' ' <i>HexEscape</i>		
' ' <i>IdentityEscape</i>		
<i>IdentityEscape</i> \square <i>NonTerminator</i> except _ <i>UnicodeAlphanumeric</i>		(UnicodeAlphanumeric: 7.5)
<i>ControlEscape</i> \square b f n r t v		
<i>ZeroEscape</i> \square 0 [lookahead { <i>ASCIIDigit</i> }]		(ASCIIDigit: 7.7)
<i>HexEscape</i> \square		
x <i>HexDigit</i> <i>HexDigit</i>		(HexDigit: 7.7)
u <i>HexDigit</i> <i>HexDigit</i> <i>HexDigit</i> <i>HexDigit</i>		

Semantics

Lex[*StringLiteral*]: TOKEN;

Lex[*StringLiteral* \square ' *StringChars*^{single} '] = a **String** token with the value LexString[*StringChars*^{single}];

Lex[*StringLiteral* \square " *StringChars*^{double} "] = a **String** token with the value LexString[*StringChars*^{double}];

LexString[*StringChars*[□]]: STRING;

LexString[*StringChars*[□] \square «empty»] = "",

LexString[*StringChars*₀[□] \square *StringChars*₁[□] *StringChar*[□]] = LexString[*StringChars*₀[□]] \oplus [LexChar[*StringChar*₁[□]]];

LexString[*StringChars*₀[□] \square *StringChars*₁[□] *NullEscape*] = LexString[*StringChars*₀[□]];

LexChar[*StringChar*[□]]: CHARACTER;

LexChar[*StringChar*[□] \square *LiteralStringChar*[□]] = *LiteralStringChar*[□];

LexChar[*StringChar*[□] \square \ *StringEscape*] = LexChar[*StringEscape*];

LexChar[*StringEscape*]: CHARACTER;

LexChar[*StringEscape* \square *ControlEscape*] = LexChar[*ControlEscape*];

LexChar[*StringEscape* \square *ZeroEscape*] = LexChar[*ZeroEscape*];

LexChar[*StringEscape* \square *HexEscape*] = LexChar[*HexEscape*];

LexChar[*StringEscape* \square *IdentityEscape*] = *IdentityEscape*;

Lex[StringLiteral] \sqsubseteq ! StringChars^{single}]
 Return a **string** token with string contents LexString[StringChars^{single}].

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

LexString[StringChars[#]] \sqsubseteq «empty» = ""

LexString[StringChars[#]] \sqsubseteq StringChars[#] + StringChar[#]
 Return a string consisting of the string LexString[StringChars[#]] concatenated with the character LexChar[StringChar[#]].

LexString[StringChars[#]] \sqsubseteq StringChars[#] + NullEscape = LexString[StringChars[#]]

LexChar[StringChar[#]] \sqsubseteq LiteralStringChar[#]
 Return the character LiteralStringChar[#].

LexChar[StringChar[#]] \sqsubseteq \ StringEscape = LexChar[StringEscape]

LexChar[StringEscape] \sqsubseteq ControlEscape = LexChar[ControlEscape]

LexChar[StringEscape] \sqsubseteq ZeroEscape = LexChar[ZeroEscape]

LexChar[StringEscape] \sqsubseteq HexEscape = LexChar[HexEscape]

LexChar[StringEscape] \sqsubseteq IdentityEscape
 Return the character IdentityEscape.

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

LexChar[ControlEscape]: CHARACTER:
LexChar[ControlEscape] \sqsubseteq b = '«BS»';
LexChar[ControlEscape] \sqsubseteq f = '«FF»';
LexChar[ControlEscape] \sqsubseteq n = '«LF»';
LexChar[ControlEscape] \sqsubseteq r = '«CR»';
LexChar[ControlEscape] \sqsubseteq t = '«TAB»';
LexChar[ControlEscape] \sqsubseteq v = '«VT»';

LexChar[ZeroEscape] \sqsubseteq 0 [lookahead {ASCIIDigit}]: CHARACTER = '«NUL»';

LexChar[HexEscape]: CHARACTER:
LexChar[HexEscape] \sqsubseteq x HexDigit₁ HexDigit₂
 = codeToCharacter(16HexValue[HexDigit₁] + HexValue[HexDigit₂]);
LexChar[HexEscape] \sqsubseteq u HexDigit₁ HexDigit₂ HexDigit₃ HexDigit₄
 = codeToCharacter(4096HexValue[HexDigit₁] + 256HexValue[HexDigit₂] + 16HexValue[HexDigit₃] + HexValue[HexDigit₄]);

LexChar[ControlEscape] \sqsubseteq b = '«BS»'
LexChar[ControlEscape] \sqsubseteq f = '«FF»'
LexChar[ControlEscape] \sqsubseteq n = '«LF»'
LexChar[ControlEscape] \sqsubseteq r = '«CR»'
LexChar[ControlEscape] \sqsubseteq t = '«TAB»'
LexChar[ControlEscape] \sqsubseteq v = '«VT»'

~~LexChar[ZeroEscape □ 0 [lookahead {ASCIIDigit}]] = «NUL»~~

~~LexChar[HexEscape □ -x HexDigit₁ HexDigit₂]
Let n = 16*LexNumber[HexDigit₁] + LexNumber[HexDigit₂].
Return the character with code point value n.~~

~~LexChar[HexEscape □ -u HexDigit₁ HexDigit₂ HexDigit₃ HexDigit₄]
Let n = 4096*LexNumber[HexDigit₁] + 256*LexNumber[HexDigit₂] + 16*LexNumber[HexDigit₃] +
LexNumber[HexDigit₄].
Return the character with code point value n.~~

NOTE A *LineTerminator* character cannot appear in a string literal, even if preceded by a backslash \. The correct way to cause a line terminator character to be part of the string value of a string literal is to use an escape sequence such as \n or \u000A.

7.9 Regular expression literals

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegExpBody* and the *RegExpFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it should not extend the *RegExpBody* and *RegExpFlags* productions or the productions used by these productions.

Syntax

RegExpLiteral □ *RegExpBody* *RegExpFlags*

RegExpFlags □
 «empty»
 | *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)

Lex[*RegExpLiteral* □ *RegExpBody* *RegExpFlags*]

Return a **regularExpression** token with the body string LexString[*RegExpBody*] and flags string

LexString[*RegExpFlags*].

LexString[*RegExpFlags* □ «empty»] = “”

LexString[*RegExpFlags* □ *RegExpFlags*₁ *ContinuingIdentifierCharacterOrEscape*]

Return a string consisting of the string LexString[*RegExpFlags*₁] concatenated with the character

LexChar[*ContinuingIdentifierCharacterOrEscape*].

LexString[*RegExpFlags* □ *RegExpFlags*₁ *NullEscape*] = LexString[*RegExpFlags*₁]

LexString[*RegExpBody* □ / [lookahead [(*)] *RegExpChars* /] = LexString[*RegExpChars*]

LexString[*RegExpChars* □ *RegExpChar*] = LexString[*RegExpChar*]

LexString[*RegExpChars* □ *RegExpChars*₁ *RegExpChar*]

Return a string consisting of the string LexString[*RegExpChars*₁] concatenated with the string

LexString[*RegExpChar*].

LexString[*RegExpChar* □ *OrdinaryRegExpChar*]

Return a string consisting of the single character *OrdinaryRegExpChar*.

LexString[*RegExpChar* □ \ NonTerminator]

Return a string consisting of the two characters ‘\’ and *NonTerminator*.

NOTE A regular expression literal is an input element that is converted to a `RegExp` object (section *****) when it is scanned. The object is created before evaluation of the containing program or function begins. Evaluation of the literal produces a reference to that object; it does not create a new object. Two regular expression literals in a program evaluate to regular expression objects that never compare as === to each other even if the two literals' contents are identical. A `RegExp` object may also be created at runtime by `new RegExp` (section *****) or calling the `RegExp` constructor as a function (section *****).

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters // start a single-line comment. To specify an empty regular expression, use /(?:)/.

8 Program Structure

8.1 Packages

8.2 Scopes

9 Data Model

This chapter describes the essential state held in various ECMAScript objects. This state is presented abstractly using the formalisms from chapter 5. Much of the state held in these objects is observable by ECMAScript programmers only

indirectly, and implementations are encouraged to implement these objects in more efficient ways as long as the observable behaviour is the same as described here.

9.1 Objects

An object is a first-class data value visible to ECMAScript programmers. Every object is either **undefined**, **null**, a Boolean, a 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 method closure, a prototype instance, a class instance, a package object, or the global object. These kinds of objects are described in the subsections below.

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

```
OBJECT = UNDEFINED □ NULL □ BOOLEAN □ LONG □ ULONG □ FLOAT32 □ FLOAT64 □ CHARACTER □ STRING □  
NAMESPACE □ COMPOUNDATTRIBUTE □ CLASS □ METHODCLOSURE □ PROTOTYPE □ INSTANCE □ PACKAGE □  
GLOBAL;
```

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

```
PRIMITIVEOBJECT  
= UNDEFINED □ NULL □ BOOLEAN □ LONG □ ULONG □ FLOAT32 □ FLOAT64 □ CHARACTER □ STRING;
```

A **DYNAMICOBJECT** is an object that can host dynamic properties:

```
DYNAMICOBJECT = PROTOTYPE □ SIMPLEINSTANCE □ CALLABLEINSTANCE □ DYNAMICINSTANCE □ GLOBAL;
```

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

```
OBJECTOPT = OBJECT □ {none};
```

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

```
OBJECTI = OBJECT □ {inaccessible};
```

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

```
OBJECTIOPT = OBJECT □ {inaccessible, none};
```

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

```
OBJECTU = OBJECT □ {uninitialised};
```

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

```
BOOLEANOPT = BOOLEAN □ {none};
```

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

```
INTEGEROPT = INTEGER □ {none};
```

9.1.1 Undefined

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

```
UNDEFINED = {undefined}
```

9.1.2 Null

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

```
NULL = {null}
```

9.1.3 Booleans

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

9.1.4 Numbers

The semantic domains **LONG**, **ULONG**, **FLOAT32**, and **FLOAT64**, collectively denoted by the domain **GENERALNUMBER**, represent the numeric types supported by ECMAScript. See section 5.12.

9.1.5 Strings

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

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

STRINGOPT = **STRING** □ {**none**}

9.1.6 Namespaces

A namespace object is represented by a **NAMESPACE** record (see section 5.11) with the field below. Each time a namespace is created, the new namespace is different from every other namespace, even if it happens to share the name of an existing namespace.

Field	Contents	Note
name	STRING	The namespace's name used by toString

9.1.6.1 Qualified Names

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

Field	Contents	Note
-------	----------	------

namespace	NAMESPACE	The namespace qualifier
------------------	------------------	-------------------------

id	STRING	The name
-----------	---------------	----------

QUALIFIEDNAMEOPT consists of all qualified names as well as **none**:

QUALIFIEDNAMEOPT = **QUALIFIEDNAME** □ {**none**}

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

MULTINAME = **QUALIFIEDNAME**{}

9.1.7 Compound attributes

Compound attribute objects are all values obtained from combining zero or more syntactic attributes (see *****) that are not Booleans or single namespaces. A compound attribute object is represented by a **COMPOUNDATTRIBUTE** tuple (see section 5.10) with the fields below.

Field	Contents	Note
-------	----------	------

namespaces	NAMESPACE {}	The set of namespaces contained in this attribute
-------------------	---------------------	---

explicit	BOOLEAN	true if the explicit attribute has been given
-----------------	----------------	---

dynamic	BOOLEAN	true if the dynamic attribute has been given
----------------	----------------	--

memberMod	MEMBERMODIFIER	static , constructor , abstract , virtual , or final if one of these attributes has been given; none if not. MEMBERMODIFIER = { none , static , constructor , abstract , virtual , final }
------------------	-----------------------	---

overrideMod	OVERRIDEMODIFIER	true , false , or undefined if the override attribute with one of these
--------------------	-------------------------	---

arguments was given; **true** if the attribute `override` without arguments was given; **none** if the `override` attribute was not given. **OVERRIDEMODIFIER** = **{none, true, false, undefined}**

<code>prototype</code>	<code>BOOLEAN</code>	true if the <code>prototype</code> attribute has been given
<code>unused</code>	<code>BOOLEAN</code>	true if the <code>unused</code> 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` \sqcup `NAMESPACE` \sqcup `COMPOUNDATTRIBUTE`

ATTRIBUTEOPTNOTFALSE consists of **none** as well as all attributes and attribute combinations except for **false**:

ATTRIBUTEOPTNOTFALSE = **{none, true}** \sqcup `NAMESPACE` \sqcup `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
<code>staticReadBindings</code>	<code>STATICBINDING</code> {}	Map of qualified names to readable static members defined in this class (see section *****)
<code>staticWriteBindings</code>	<code>STATICBINDING</code> {}	Map of qualified names to writable static members defined in this class
<code>instanceReadBindings</code>	<code>INSTANCEBINDING</code> {}	Map of qualified names to readable instance members defined in this class
<code>instanceWriteBindings</code>	<code>INSTANCEBINDING</code> {}	Map of qualified names to writable instance members defined in this class
<code>instanceInitOrder</code>	<code>INSTANCEVARIABLE</code> []	List of instance variables defined in this class in the order in which they are initialised
<code>complete</code>	<code>BOOLEAN</code>	true after all members of this class have been added to this CLASS record
<code>super</code>	<code>CLASSOPT</code>	This class's immediate superclass or null if none
<code>prototype</code>	<code>OBJECT</code>	An object that serves as this class's prototype for compatibility with ECMAScript 3; may be null
<code>privateNamespace</code>	<code>NAMESPACE</code>	This class's <code>private</code> namespace
<code>dynamic</code>	<code>BOOLEAN</code>	true if this class or any of its ancestors was defined with the <code>dynamic</code> attribute
<code>allowNull</code>	<code>BOOLEAN</code>	true if null is considered to be an instance of this class
<code>final</code>	<code>BOOLEAN</code>	true if this class cannot be subclassed
<code>call</code>	<code>OBJECT</code> \sqcup <code>ARGUMENTLIST</code> \sqcup <code>PHASE</code> \sqcup <code>OBJECT</code>	A procedure to call (see section 9.5) when this class is used in a call expression
<code>construct</code>	<code>ARGUMENTLIST</code> \sqcup <code>PHASE</code> \sqcup <code>OBJECT</code>	A procedure to call (see section 9.5) when this class is used in a <code>new</code> expression
<code>implicitCoerce</code>	<code>OBJECT</code> \sqcup <code>OBJECT</code>	A procedure to call when a value is assigned to a variable, parameter, or result whose type is this class. The argument to <code>implicitCoerce</code> 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, <code>implicitCoerce</code> should throw an

defaultValueOBJECTexception.

When a variable whose type is this class is defined but not explicitly initialised, the variable's initial value is defaultValue, which must be an instance of this class.

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

CLASSOPT = **CLASS** □ {**none**}

A **CLASS** *c* is an *ancestor* of **CLASS** *d* if either *c* = *d* or *d.super* = *s*, *s* ≠ **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 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	<u>OBJECT</u>	The bound this value
method	<u>INSTANCEMETHOD</u>	The bound method

9.1.10 Prototype Instances

Prototype instances are represented as **PROTOTYPE** records (see section 5.11) with the fields below. Prototype instances contain no fixed properties.

Field	Contents	Note
parent	<u>PROTOTYPEOPT</u>	If this instance was created by calling new on a prototype function, the value of the function's prototype property at the time of the call; none otherwise.
dynamicProperties	<u>DYNAMICPROPERTY</u> {}	A set of this instance's dynamic properties

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

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

A **DYNAMICPROPERTY** record (see section 5.11) has the fields below and describes one dynamic property of one (prototype or class) instance.

Field	Contents	Note
name	<u>STRING</u>	This dynamic property's name
value	<u>OBJECT</u>	This dynamic property's current value

9.1.11 Class Instances

Instances of programmer-defined classes as well as of some built-in classes have the semantic domain **INSTANCE**. If the class of an instance or one of its ancestors has the dynamic attribute instance responds to the function call or new operators, then the that instance is a CALLABLEINSTANCEDYNAMICINSTANCE record; otherwise, it is a SIMPLEINSTANCEFIXEDINSTANCE record. An instance can also be an **ALIASINSTANCE** that refers to another instance. This specification uses **ALIASINSTANCES** to permit but not require an implementation to share function closures with identical behaviour.

INSTANCE = **SIMPLEINSTANCE** □ **CALLABLEINSTANCE** □ **ALIASINSTANCE**
INSTANCE = **NONALIASINSTANCE** □ **ALIASINSTANCE**
NEE;

~~NONALIASINSTANCE = FIXEDINSTANCE □ DYNAMICINSTANCE;~~

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

~~Instances of non dynamic classes are represented as **FIXEDINSTANCE** records (see section 5.11) with the fields below. These instances can contain only fixed properties.~~

Field	Contents	Note
type	CLASS	This instance's type
call	OBJECT □ ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT	A procedure to call when this instance is used in a call expression. The procedure takes an OBJECT (the this value), an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
construct	ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT	A procedure to call when this instance is used in a new expression. The procedure takes an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
env	ENVIRONMENT	The environment to pass to the call or construct procedure
typeofString	STRING	A string to return if typeof is invoked on this instance
slots	SLOT{}	A set of slots that hold this instance's fixed property values

~~Instances which do not respond to the function **call** or **new** operators are represented as **SIMPLEINSTANCE** records (see section 5.11) with the fields below.~~

Field	Contents	Note
type	CLASS	This instance's type
typeofString	STRING	A string to return if typeof is invoked on this instance
slots	SLOT{}	A set of slots that hold this instance's fixed property values
dynamicProperties	DYNAMICPROPERTY{} □ {fixed}	A set of this instance's dynamic properties if this instance's class is a dynamic class; fixed if not

~~Instances of dynamic classes which respond to the function **call** or **new** operators are represented as **CALLABLEINSTANCE** **DYNAMICINSTANCE** records (see section 5.11) with the fields below. These instances can contain fixed and dynamic properties.~~

Field	Contents	Note
type	CLASS	This instance's type
typeofString	STRING	A string to return if typeof is invoked on this instance
slots	SLOT{}	A set of slots that hold this instance's fixed property values
dynamicProperties	DYNAMICPROPERTY{} □ {fixed}	A set of this instance's dynamic properties if this instance's class is a dynamic class; fixed if not
call	OBJECT □ ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT	A procedure to call when this instance is used in a call expression. The procedure takes an OBJECT (the this value), an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result
construct	ARGUMENTLIST □ ENVIRONMENT □ PHASE □ OBJECT	A procedure to call when this instance is used in a new expression. The procedure takes an ARGUMENTLIST (see section 9.5), a lexical ENVIRONMENT and a PHASE (see

section 9.5), a lexical ENVIRONMENT, and a PHASE (see section 9.6) and produces an OBJECT result

env	ENVIRONMENT	The environment to pass to the call or construct procedure
-----	-------------	--

ALIASINSTANCE records (see section 5.11) with the fields below represent aliases to existing instances. An ALIASINSTANCE behaves just like its original instance except that it supplies a different environment to the **call** and **construct** procedures. In practice, an implementation would likely only use ALIASINSTANCEs if it can prove that supplying the different environment to the **call** and **construct** procedures has no visible consequences, so it could optimise out the ALIASINSTANCE altogether.

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

9.11.1 Open Instances

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

Field	Contents	Note
<u>type</u>	<u>CLASS</u>	<u>Values to be transferred into the generated CALLABLEINSTANCE's corresponding fields</u>
<u>typeofString</u>	<u>STRING</u>	
<u>defaultSlots</u> <u>instantiate</u>	<u>SLOT</u> { } <u>ENVIRONMENT</u> □ <u>NONALIASINSTANCE</u>	<u>A procedure to call to supply an environment and obtain a fresh instanceA list of the default values of the generated CALLABLEINSTANCE's slots</u>
<u>buildPrototype</u>	<u>BOOLEAN</u>	<u>If true, the generated CALLABLEINSTANCE gets a separate prototype slot with its own prototype object</u>
<u>call</u>	<u>OBJECT</u> □ <u>ARGUMENTLIST</u> □ <u>ENVIRONMENT</u> □ <u>PHASE</u> □ <u>OBJECT</u>	<u>Values to be transferred into the generated CALLABLEINSTANCE's corresponding fields</u>
<u>construct</u>	<u>ARGUMENTLIST</u> □ <u>ENVIRONMENT</u> □ <u>PHASE</u> □ <u>OBJECT</u>	
cache	<u>CALLABLEINSTANCE</u> <u>NONALIASINSTANCE</u> □ {none}	Optional cached value of the last instantiation. This cache serves only to precisely specify the closure sharing optimization and would likely not be present in any actual implementation.

9.11.2 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
<u>id</u>	<u>INSTANCEVARIABLE</u>	The instance variable whose value this slot carries
<u>value</u>	<u>OBJECT</u>	This fixed property's current value; uninitialised if the fixed property is an uninitialised constant

9.1.12 Packages

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

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

9.1.13 Global Objects

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

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

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	INSTANCE	The value of <code>expr</code> to which the <code>super</code> subexpression was applied; if <code>expr</code> wasn't given, defaults to the value of <code>this</code> . The value of <code>instance</code> is always an instance of the <code>limit</code> class or one of its descendants.
limit	CLASS	The class inside which the <code>super</code> subexpression was applied

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

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

OBJOPTIONALLIMIT = **OBJECT** □ **LIMITEDINSTANCE**

9.3 References

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

REFERENCE = **LEXICALREFERENCE** □ **DOTREFERENCE** □ **BRACKETREFERENCE**;

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

OBJORREF = **OBJECT** □ **REFERENCE**

A **LEXICALREFERENCE** tuple (see section 5.10) has the fields below and represents an lvalue that refers to a variable with one of a given set of qualified names. **LEXICALREFERENCE** tuples arise from evaluating identifiers `a` and qualified identifiers `q :: a`.

Field	Contents	Note
-------	----------	------

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

A **DOTREFERENCE** tuple (see section 5.10) has the fields below and represents an lvalue that refers to a property of the base object with one of a given set of qualified names. **DOTREFERENCE** tuples arise from evaluating subexpressions such as *a.b* or *a.q::b*.

Field	Contents	Note
base	OBJOPTIONALLIMIT	The object whose property was referenced (<i>a</i> in the examples above). The object may be a LIMITEDINSTANCE if <i>a</i> is a super expression, in which case the property lookup will be restricted to members defined in proper ancestors of base.limit .
propertyMultiname	MULTINAME	A nonempty set of qualified names to which this reference can refer (<i>b</i> qualified with the namespace <i>q</i> or all currently open namespaces in the example above)

A **BRACKETREFERENCE** tuple (see section 5.10) has the fields below and represents an lvalue that refers to the result of applying the **[]** operator to the base object with the given arguments. **BRACKETREFERENCE** tuples arise from evaluating subexpressions such as *a[x]* or *a[x,y]*.

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

9.4 Function Support

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

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

A **SIGNATURE** tuple (see section 5.10) has the fields below and represents the type signature of a function.

Field	Contents	Note
positional	PARAMETER[]	List of the required positional parameters
optionalPositional	PARAMETER[]	List of the optional positional parameters, which follow the required positional parameters
optionalNamed	NAMEDPARAMETER{}	Set of the types and names of the optional named parameters
rest	PARAMETER \sqcup { none }	The parameter for collecting any extra arguments that may be passed or null if no extra arguments are allowed
restAllowsNames	BOOLEAN	true if the extra arguments may be named
returnType	CLASS	The type of this function's result

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

Field	Contents	Note
localName	QUALIFIEDNAMEOPT	Name of the local variable that will hold this parameter's value

type	CLASS	This parameter's type
-------------	--------------	-----------------------

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

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

9.5 Argument Lists

An **ARGUMENTLIST** tuple (see section 5.10) has the fields below and describes the arguments (other than **this**) passed to a function.

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

A **NAMEDARGUMENT** tuple (see section 5.10) has the fields below and describes one named argument passed to a function.

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

9.6 Modes of expression evaluation

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

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

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

9.7 Contexts

A **CONTEXT** tuple (see section 5.10) carries static information about a particular point in the source program and has the fields below.

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

9.8 Labels

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

LABEL = **STRING** □ {**default**}

A **JUMPTARGETS** tuple (see section 5.10) describes the sets of labels that are valid destinations for **break** or **continue** statements at a point in the source code. A **JUMPTARGETS** tuple has the fields below.

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

9.9 Environments

Environments contain the bindings that are visible from a given point in the source code. An **ENVIRONMENT** is a list of two or more frames. Each frame corresponds to a scope. More specific frames are listed first—each frame’s scope is directly contained in the following frame’s scope. The last frame is always the **SYSTEMFRAME**. The next-to-last frame is always a **PACKAGE** or **GLOBAL** frame.

ENVIRONMENT = FRAME[]

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

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

9.9.1 Frames

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

FRAME = **SYSTEMFRAME** □ **GLOBAL** □ **PACKAGE** □ **PARAMETERFRAME** □ **CLASS** □ **BLOCKFRAME**;

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

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

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

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

9.9.1.1 System Frame

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

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

9.9.1.2 Function Parameter Frames

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

Field	Contents	Note
staticReadBindings	STATICBINDING{}	Map of qualified names to readable definitions in this function
staticWriteBindings	STATICBINDING{}	Map of qualified names to writable definitions in this function
plurality	PLURALITY	See section 9.9.1
this	OBJECTOPT	The value of <code>this</code> ; <code>none</code> if this function doesn’t define <code>this</code> ;

		inaccessible if this function defines <code>this</code> but the value is not available because this function hasn't been called yet
prototype	BOOLEAN	true if this function is not an instance method but defines <code>this</code> anyway

9.9.1.3 Block Frames

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

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

9.9.2 Static Bindings

A **STATICBINDING** tuple (see section 5.10) has the fields below and describes the member to which one qualified name is bound in a frame. Multiple qualified names may be bound to the same member in a frame, but a qualified name may not be bound to multiple members in a frame (except when one binding is for reading only and the other binding is for writing only).

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

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

STATICMEMBER = {**forbidden**} □ **VARIABLE** □ **HOISTEDVAR** □ **CONSTRUCTORMETHOD** □ **GETTER** □ **SETTER**;

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

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

A **VARIABLE** record (see section 5.11) has the fields below and describes one variable or constant definition.

Field	Contents	Note
<code>type</code>	VARIABLETYPE	Type of values that may be stored in this variable (see below)
<code>value</code>	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
<code>immutable</code>	BOOLEAN	true if this variable's value may not be changed once set

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

VARIABLETYPE = **CLASS** □ {**inaccessible**} □ () □ **CLASS**

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

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

A **HOISTEDVAR** record (see section 5.11) has the fields below and describes one hoisted variable.

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

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

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

A `GETTER` record (see section 5.11) has the fields below and describes one static getter definition.

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

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

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

9.9.3 Instance Bindings

An `INSTANCEBINDING` tuple (see section 5.10) has the fields below and describes the binding of one qualified name to an instance member of a class. Multiple qualified names may be bound to the same instance member in a class, but a qualified name may not be bound to multiple instance members in a class (except when one binding is for reading only and the other binding is for writing only).

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

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

`INSTANCEMEMBER = INSTANCEVARIABLE □ INSTANCEMETHOD □ INSTANCEGETTER □ INSTANCESETTER;`

`INSTANCEMEMBEROPT = INSTANCEMEMBER □ {none};`

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

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

final	BOOLEAN	true if this member may not be overridden in subclasses
--------------	----------------	--

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

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

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

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

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

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

10 Data Operations

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

10.1 Numeric Utilities

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

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

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

```
proc signedWrap32(i: INTEGER): {- $2^{31}$  ...  $2^{31}-1$ }  
    j: INTEGER  $\sqcup$  bitwiseAnd(i, 0xFFFFFFFF);  
    if j  $\geq 2^{31}$  then j  $\sqcup$  j -  $2^{32}$  end if;  
    return j  
end proc;
```

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

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

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

```
proc signedWrap64(i: INTEGER): {- $2^{63}$  ...  $2^{63}-1$ }
    j: INTEGER □ bitwiseAnd(i, 0xFFFFFFFFFFFFFFFFF);
    if j ≥  $2^{63}$  then j □ j -  $2^{64}$  end if;
    return j
end proc;
```

```
proc truncateToInteger(x: GENERALNUMBER): INTEGER
    case x of
        {+∞f32, +∞f64, -∞f32, -∞f64, NaN32f32, NaN64f64} do return 0;
        FINITEFLOAT32 do return truncateFiniteFloat32(x);
        FINITEFLOAT64 do return truncateFiniteFloat64(x);
        LONG □ ULONG do return x.value
    end case
end proc;
```

```
proc checkInteger(x: GENERALNUMBER): INTEGEROPT
    case x of
        {NaN32f32, NaN64f64, +∞f32, +∞f64, -∞f32, -∞f64} do return none;
        {+zerof32, +zerof64, -zerof32, -zerof64} do return 0;
        LONG □ ULONG do return x.value;
        NONZEROFINITEFLOAT32 □ NONZEROFINITEFLOAT64 do
            r: RATIONAL □ x.value;
            if r □ INTEGER then return none end if;
            return r
    end case
end proc;
```

```
proc integerToLong(i: INTEGER): GENERALNUMBER
    if  $-2^{63} \leq i \leq 2^{63}-1$  then return LONG[value: i]
    elseif  $2^{63} \leq i \leq 2^{64}-1$  then return ULONG[value: i]
    else return realToFloat64(i)
    end if
end proc;
```

```
proc integerToULong(i: INTEGER): GENERALNUMBER
    if  $0 \leq i \leq 2^{64}-1$  then return ULONG[value: i]
    elseif  $-2^{63} \leq i \leq -1$  then return LONG[value: i]
    else return realToFloat64(i)
    end if
end proc;
```

```
proc rationalToLong(q: RATIONAL): GENERALNUMBER
    if q □ INTEGER then return integerToLong(q)
    elseif |q| ≤  $2^{53}$  then return realToFloat64(q)
    elseif q <  $-2^{63}-1/2$  or q ≥  $2^{64}-1/2$  then return realToFloat64(q)
    else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        Note that  $-2^{63} \leq i \leq 2^{64}-1$ 
        if i <  $2^{63}$  then return LONG[value: i] else return ULONG[value: i] end if
    end if
end proc;
```

```

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

proc toRational(x: FINITEGENERALNUMBER): RATIONAL
  case x of
    {+zerof32, +zerof64, -zerof32, -zerof64} do return 0;
    NONZEROFINITEFLOAT32  $\sqsubseteq$  NONZEROFINITEFLOAT64  $\sqsubseteq$  LONG  $\sqsubseteq$  ULONG do return x.value
  end case
end proc;

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

```

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

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

```

proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
  if x  $\sqsubseteq$  {NaN32f32, NaN64f64} or y  $\sqsubseteq$  {NaN32f32, NaN64f64} then return unordered
  elsif x  $\sqsubseteq$  {+∞f32, +∞f64} and y  $\sqsubseteq$  {+∞f32, +∞f64} then return equal
  elsif x  $\sqsubseteq$  {-∞f32, -∞f64} and y  $\sqsubseteq$  {-∞f32, -∞f64} then return equal
  elsif x  $\sqsubseteq$  {+∞f32, +∞f64} or y  $\sqsubseteq$  {-∞f32, -∞f64} then return greater
  elsif x  $\sqsubseteq$  {-∞f32, -∞f64} or y  $\sqsubseteq$  {+∞f32, +∞f64} then return less
  else
    xr: RATIONAL  $\sqsubseteq$  toRational(x);
    yr: RATIONAL  $\sqsubseteq$  toRational(y);
    if xr < yr then return less
    elsif xr > yr then return greater
    else return equal
    end if
  end if
end proc;

```

10.2 Object Utilities

```

proc resolveAlias(o: INSTANCE): SIMPLEINSTANCE  $\sqsubseteq$  CALLABLEINSTANCE
  case o of
    ALIASINSTANCE do return o.original;
    SIMPLEINSTANCE  $\sqsubseteq$  CALLABLEINSTANCE do return o
  end case
end proc;

```

10.2.1 objectType

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

```

proc objectType(o: OBJECT): CLASS
  case o of
    UNDEFINED do return undefinedClass;
    NULL do return nullClass;
    BOOLEAN do return booleanClass;
    LONG do return longClass;
    ULONG do return uLongClass;
    FLOAT32 do return floatClass;
    FLOAT64 do return numberClass;
    CHARACTER do return characterClass;
    STRING do return stringClass;
    NAMESPACE do return namespaceClass;
    COMPOUNDATTRIBUTE do return attributeClass;
    CLASS do return classClass;
    METHODCLOSURE do return functionClass;
    PROTOTYPE do return prototypeClass;
    INSTANCE do return resolveAlias(o).type;
    PACKAGE GLOBAL do return packageClass
  end case
end proc;

```

10.2.2 *hasType*

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

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

```

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

```

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

```

proc relaxedHasType(o: OBJECT, c: CLASS): BOOLEAN
    t: CLASS  $\sqsubseteq$  objectType(o);
    return isAncestor(c, t) or (o = null and c.allowNull)
end proc;

```

10.2.3 *toBoolean*

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

```

proc toBoolean(o: OBJECT, phase: PHASE): BOOLEAN
  case o of
    UNDEFINED [] NULL do return false;
    BOOLEAN do return o;
    LONG [] ULONG do return o.value ≠ 0;
    FLOAT32 do return o [] {+zerof32, -zerof32, NaN32f32};
    FLOAT64 do return o [] {+zerof64, -zerof64, NaN64f64};
    STRING do return o ≠ "";
    CHARACTER [] NAMESPACE [] COMPOUNDATTRIBUTE [] CLASS [] METHODCLOSURE [] PROTOTYPE [] INSTANCE []
      PACKAGE [] GLOBAL do
        return true
    end case
  end proc;

```

10.2.4 *toGeneralNumber*

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

```
proc toGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
  case o of
    UNDEFINED do return NaN64f64;
    NULL [] {false} do return +zerof64;
    {true} do return 1.0f64;
    GENERALNUMBER do return o;
    CHARACTER [] STRING do ???;
    NAMESPACE [] COMPOUNDATTRIBUTE [] CLASS [] METHODCLOSURE [] PACKAGE [] GLOBAL do
      throw badValueError;
    PROTOTYPE [] INSTANCE do ????
  end case
end proc;
```

10.2.5 *toString*

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

```
proc toString(o: OBJECT, phase: PHASE): STRING
  case o of
    UNDEFINED do return "undefined";
    NULL do return "null";
    {false} do return "false";
    {true} do return "true";
    LONG [] ULONG do return integerToString(o.value);
    FLOAT32 do return float32ToString(o);
    FLOAT64 do return float64ToString(o);
    CHARACTER do return [o];
    STRING do return o;
    NAMESPACE do ???;
    COMPOUNDATTRIBUTE do ???;
    CLASS do ???;
    METHODCLOSURE do ???;
    PROTOTYPE [] INSTANCE do ???;
    PACKAGE [] GLOBAL do ???
  end case
end proc;
```

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

```
proc integerToString(i: INTEGER): STRING
  if i < 0 then return [‘-’] ⊕ integerToString(-i) end if;
  q: INTEGER [] i/10[];
  r: INTEGER [] i - q[10];
  c: CHARACTER [] codeToCharacter(r + characterToCode(‘0’));
  if q = 0 then return [c] else return integerToString(q) ⊕ [c] end if
end proc;
```

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

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

`float32ToString(x)` converts a `FLOAT32` x to a string using fixed-point notation if the absolute value of x is between 10^{-6} inclusive and 10^{21} exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a `FLOAT32` value would result in the same value x (except that `-zerof32` would become `+zerof32`).

```
proc float32ToString(x: FLOAT32): STRING
  case x of
    {NaN32f32} do return "NaN";
    {+zerof32, -zerof32} do return "0";
    {+∞f32} do return "Infinity";
    {-∞f32} do return "-Infinity";
    NONZEROFINITEFLOAT32 do
      r: RATIONAL ⊑ x.value;
      if r < 0 then return "-" ⊕ float32ToString(float32Negate(x))
      else
        Let n, k, and s be integers such that  $k \geq 1$ ,  $10^{k-1} \leq s \leq 10^k$ ,  $\text{realToFloat32}(s \square 10^{n-k}) = x$ , and  $k$  is as small as possible. Note that  $k$  is the number of digits in the decimal representation of  $s$ , that  $s$  is not divisible by 10, and that the least significant digit of  $s$  is not necessarily uniquely determined by these 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  $s \square 10^{n-k}$  is closest in value to  $r$ ; if there are two such possible values of  $s$ , choose the one that is even.
        digits: STRING ⊑ integerToString(s);
        if k ≤ n ≤ 21 then return digits ⊕ repeat('0', n - k)
        elseif 0 < n ≤ 21 then return digits[0 ... n - 1] ⊕ "." ⊕ digits[n ...]
        elseif -6 < n ≤ 0 then return "0." ⊕ repeat('0', -n) ⊕ digits
        else
          mantissa: STRING;
          if k = 1 then mantissa ⊑ digits
          else mantissa ⊑ digits[0 ... 0] ⊕ "." ⊕ digits[1 ...]
          end if;
          return mantissa ⊕ "e" ⊕ integerToStringWithSign(n - 1)
        end if
      end if
    end case
  end proc;
```

`float64ToString(x)` converts a `FLOAT64` x to a string using fixed-point notation if the absolute value of x is between 10^{-6} inclusive and 10^{21} exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a `FLOAT64` value would result in the same value x (except that `-zerof64` would become `+zerof64`).

```

proc float64ToString(x: FLOAT64): STRING
  case x of
    {NaN164} do return “NaN”;
    {+zero164, -zero164} do return “0”;
    {+∞164} do return “Infinity”;
    {-∞164} do return “-Infinity”;
    NONZEROFINITEFLOAT64 do
      r: RATIONAL ┌ x.value;
      if r < 0 then return “-” ┌ float64ToString(float64Negate(x))
      else
        Let n, k, and s be integers such that k ≥ 1,  $10^{k-1} \leq s \leq 10^k$ , realToFloat64(s) · 10n-k = x, and k is as small as
        possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by
        10, and that the least significant digit of s is not necessarily uniquely determined by these 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 · 10n-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 ≤ 21 then return digits ┌ repeat('0', n - k)
        elseif 0 < n ≤ 21 then return digits[0 ... n - 1] ┌ “.” ┌ digits[n ...]
        elsif -6 < n ≤ 0 then return “0.” ┌ repeat('0', -n) ┌ digits
        else
          mantissa: STRING;
          if k = 1 then mantissa ┌ digits
          else mantissa ┌ digits[0 ... 0] ┌ “.” ┌ digits[1 ...]
          end if;
          return mantissa ┌ “e” ┌ integerToStringWithSign(n - 1)
        end if
      end if
    end case
  end proc;

```

10.2.6 *toPrimitive*

```

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

```

10.2.7 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
  elseif a ⊑ {none, true} then return b
  elseif b = true then return a
  elseif a ⊑ NAMESPACE then
    if a = b then return a
    elseif b ⊑ NAMESPACE then
      return COMPOUNDATTRIBUTE[namespaces: {a, b}, explicit: false, dynamic: false, memberMod: none,
                                overrideMod: none, prototype: false, unused: false]
    else return COMPOUNDATTRIBUTE[namespaces: b.namespaces ⊔ {a}, other fields from b]
    end if
  elseif b ⊑ NAMESPACE then
    return COMPOUNDATTRIBUTE[namespaces: a.namespaces ⊔ {b}, other fields from a]
  else
    Both a and b are compound attributes. Ensure that they have no conflicting contents.
    if (a.memberMod ≠ none and b.memberMod ≠ none and a.memberMod ≠ b.memberMod) or
      (a.overrideMod ≠ none and b.overrideMod ≠ none and a.overrideMod ≠ b.overrideMod) then
        throw badValueError
    else
      return COMPOUNDATTRIBUTE[namespaces: a.namespaces ⊔ b.namespaces,
                                explicit: a.explicit or b.explicit, dynamic: a.dynamic or b.dynamic,
                                memberMod: a.memberMod ≠ none ? a.memberMod : b.memberMod,
                                overrideMod: a.overrideMod ≠ none ? a.overrideMod : b.overrideMod,
                                prototype: a.prototype or b.prototype, unused: a.unused or b.unused]
    end if
  end if
end proc;

```

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

```

proc toCompoundAttribute(a: ATTRIBUTEOPTNOTFALSE): COMPOUNDATTRIBUTE
  case a of
    {none, true} do
      return COMPOUNDATTRIBUTE[namespaces: {}, explicit: false, dynamic: false, memberMod: none,
                                overrideMod: none, prototype: false, unused: false]
    NAMESPACE do
      return COMPOUNDATTRIBUTE[namespaces: {a}, explicit: false, dynamic: false, memberMod: none,
                                overrideMod: none, prototype: false, unused: false]
    COMPOUNDATTRIBUTE do return a
  end case
end proc;

```

10.3 References

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

```

proc readReference(r: OBJORREF, phase: PHASE): OBJECT
  case r of
    OBJECT do return r;
    LEXICALREFERENCE do return lexicalRead(r.env, r.variableMultiname, phase);
    DOTREFERENCE do
      result: OBJECTOPT ⊑ readProperty(r.base, r.propertyMultiname, propertyLookup, phase);
      if result ≠ none then return result else throw propertyAccessError end if;
    BRACKETREFERENCE do return bracketRead(r.base, r.args, phase)
  end case
end proc;

```

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

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

```
proc writeReference(r: OBJORREF, newValue: OBJECT, phase: {run})
  result: {none, ok};
  case r of
    OBJECT do throw referenceError;
    LEXICALREFERENCE do
      lexicalWrite(r.env, r.variableMultiname, newValue, not r.strict, phase);
      return;
    DOTREFERENCE do
      result □ writeProperty(r.base, r.propertyMultiname, propertyLookup, true, newValue, phase);
    BRACKETREFERENCE do result □ bracketWrite(r.base, r.args, newValue, phase)
  end case;
  if result = none then throw propertyAccessError end if
end proc;
```



```
proc bracketWrite(a: OBJOPTIONALLIMIT, args: ARGUMENTLIST, newValue: OBJECT, phase: PHASE): {none, ok}
  if phase = compile then throw compileExpressionError end if;
  if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
  name: STRING □ toString(args.positional[0], phase);
  return writeProperty(a, {QUALIFIEDNAME[namespace: publicNamespace, id: name]}, propertyLookup, true,
    newValue, phase)
end proc;
```

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

```
proc deleteReference(r: OBJORREF, strict: BOOLEAN, phase: {run}): BOOLEAN
  result: BOOLEANOPT;
  case r of
    OBJECT do if strict then throw referenceError else return true end if;
    LEXICALREFERENCE do return lexicalDelete(r.env, r.variableMultiname, phase);
    DOTREFERENCE do
      result □ deleteProperty(r.base, r.propertyMultiname, propertyLookup, phase);
    BRACKETREFERENCE do result □ bracketDelete(r.base, r.args, phase)
  end case;
  if result ≠ none then return result else return true end if
end proc;
```



```
proc bracketDelete(a: OBJOPTIONALLIMIT, args: ARGUMENTLIST, phase: PHASE): BOOLEANOPT
  if phase = compile then throw compileExpressionError end if;
  if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
  name: STRING □ toString(args.positional[0], phase);
  return deleteProperty(a, {QUALIFIEDNAME[namespace: publicNamespace, id: name]}, propertyLookup, phase)
end proc;
```

10.4 Slots

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

10.5 Environments

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

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

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

```
proc getRegionalEnvironment(env: ENVIRONMENT): FRAME[]
  i: INTEGER ⊑ 0;
  while env[i] ⊑ BLOCKFRAME do i ⊑ i + 1 end while;
  if i ≠ 0 and env[i] ⊑ CLASS then i ⊑ i - 1 end if;
  return env[0 ... i]
end proc;
```

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

```
proc getRegionalFrame(env: ENVIRONMENT): FRAME
  regionalEnv: FRAME[] ⊑ getRegionalEnvironment(env);
  return regionalEnv[|regionalEnv| - 1]
end proc;
```

```
proc getPackageOrGlobalFrame(env: ENVIRONMENT): PACKAGE ⊑ GLOBAL
  g: FRAME ⊑ env[|env| - 2];
  The penultimate frame g is always a PACKAGE or GLOBAL frame.
  return g
end proc;
```

10.5.1 Access Utilities

```
tag read;
tag write;
tag ReadWrite;
ACCESS = {read, write, ReadWrite};
```

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

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

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

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

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

```
proc addStaticBindings(f: FRAME, access: ACCESS, newBindings: STATICBINDING{})
  if access  $\sqsubseteq$  {read, readWrite} then
    f.staticReadBindings  $\sqcup$  f.staticReadBindings  $\sqcup$  newBindings
  end if;
  if access  $\sqsubseteq$  {write, readWrite} then
    f.staticWriteBindings  $\sqcup$  f.staticWriteBindings  $\sqcup$  newBindings
  end if
end proc;
```

10.5.2 Adding Static Definitions

```

proc defineStaticMember(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE{},  

    overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: STATICMEMBER): MULTINAME  

    localFrame: FRAME [] env[0];  

    if overrideMod ≠ none or (explicit and localFrame [] PACKAGE) then  

        throw definitionError  

    end if;  

    namespaces2: NAMESPACE{} [] namespaces;  

    if namespaces2 = {} then namespaces2 [] {publicNamespace} end if;  

    multiname: MULTINAME [] {QUALIFIEDNAME[namespace: ns, id: id] [] ns [] namespaces2};  

    regionalEnv: FRAME[] [] getRegionalEnvironment(env);  

    regionalFrame: FRAME [] regionalEnv[|regionalEnv| - 1];  

    if some b [] staticBindingsWithAccess(localFrame, access) satisfies b.qname [] multiname then  

        throw definitionError  

    end if;  

    for each frame [] regionalEnv[1 ...] do  

        if some b [] staticBindingsWithAccess(frame, access) satisfies  

            b.qname [] multiname and b.content ≠ forbidden then  

            throw definitionError  

        end if  

    end for each;  

    if regionalFrame [] GLOBAL and (some dp [] regionalFrame.dynamicProperties satisfies  

        QUALIFIEDNAME[namespace: publicNamespace, id: dp.name] [] multiname) then  

        throw definitionError  

    end if;  

    newBindings: STATICBINDING{} [] {STATICBINDING[qname: qname, content: m, explicit: explicit] []  

        qname [] multiname};  

    addStaticBindings(localFrame, access, newBindings);  

    Mark the bindings of multiname as forbidden in all non-innermost frames in the current region if they haven't been  

    marked as such already.  

    newForbiddenBindings: STATICBINDING{} [] {STATICBINDING[qname: qname, content: forbidden, explicit: true] []  

        qname [] multiname};  

    for each frame [] regionalEnv[1 ...] do  

        addStaticBindings(frame, access, newForbiddenBindings)  

    end for each;  

    return multiname  

end proc;

```

`defineHoistedVar(env, id, initialValue)` defines a hoisted variable with the name *id* in the environment *env*. Hoisted variables are hoisted to the global or enclosing function scope. Multiple hoisted variables may be defined in the same scope, but they may not coexist with non-hoisted variables with the same name. A hoisted variable can be defined using either a `var` or a `function` statement. If it is defined using `var`, then *initialValue* is always `undefined` (if the `var` statement has an initialiser, then the variable's value will be written later when the `var` statement is executed). If it is defined using `function`, then *initialValue* must be a function instance or open instance. According to rules inherited from ECMAScript Edition 3, if there are multiple definitions of a hoisted variable, then the initial value of that variable is `undefined` if none of the definitions is a `function` definition; otherwise, the initial value is the last `function` definition.

```

proc defineHoistedVar(env: ENVIRONMENT, id: STRING, initialValue: {undefined} □ INSTANCE □ OPENINSTANCE):
  HOISTEDVAR
  qname: QUALIFIEDNAME □ QUALIFIEDNAME □ namespace: publicNamespace, id: id □
  regionalEnv: FRAME[] □ getRegionalEnvironment(env);
  regionalFrame: FRAME □ regionalEnv[regionalEnv - 1];
  env is either the GLOBAL frame or a PARAMETERFRAME because hoisting only occurs into global or function scope.
  existingBindings: STATICBINDING{} □ {b | □ b □ staticBindingsWithAccess(regionalFrame, readWrite) such that
    b.qname = qname};
  if existingBindings = {} then
    case regionalFrame of
      GLOBAL do
        if some dp □ regionalFrame.dynamicProperties satisfies dp.name = id then
          throw definitionError
        end if;
      PARAMETERFRAME do
        if |regionalEnv| ≥ 2 then
          regionalFrame □ regionalEnv[|regionalEnv| - 2];
          existingBindings □ {b | □ b □ staticBindingsWithAccess(regionalFrame, readWrite) such that
            b.qname = qname}
        end if
      end case
    end if;
  if existingBindings = {} then
    v: HOISTEDVAR □ new HOISTEDVAR[value: initialValue, hasFunctionInitialiser: initialValue ≠ undefined] □
    addStaticBindings(regionalFrame, readWrite, {STATICBINDING[qname: qname, content: v, explicit: false]});
    return v
  elsif |existingBindings| ≠ 1 then throw definitionError
  else
    b: STATICBINDING □ the one element of existingBindings;
    m: STATICMEMBER □ b.content;
    if m □ HOISTEDVAR then throw definitionError end if;
    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;
      m.hasFunctionInitialiser □ true
    end if;
    return m
  end if
end proc;

```

10.5.3 Adding Instance Definitions

```

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

tag potentialConflict;

OVERRIDDENMEMBER = INSTANCEMEMBER □ {none, potentialConflict};

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

```

```

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

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

```

```

proc defineInstanceMember(c: CLASS, ctxt: CONTEXT, id: STRING, namespaces: NAMESPACE{},  

  overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: INSTANCEMEMBER):  

  OVERRIDESTATUSPAIR  

if explicit then throw definitionError end if;  

expectMethod: BOOLEAN  $\sqcap$  m  $\sqcap$  INSTANCEMETHOD;  

readStatus: OVERRIDESTATUS  $\sqcap$  access  $\sqcap$  {read, readWrite} ?  

  resolveOverrides(c, ctxt, id, namespaces, read, expectMethod):  

  OVERRIDESTATUS[overriddenMember: none, multiname: {}]  

writeStatus: OVERRIDESTATUS  $\sqcap$  access  $\sqcap$  {write, readWrite} ?  

  resolveOverrides(c, ctxt, id, namespaces, write, expectMethod):  

  OVERRIDESTATUS[overriddenMember: none, multiname: {}]  

if readStatus.overriddenMember  $\sqcap$  INSTANCEMEMBER or  

  writeStatus.overriddenMember  $\sqcap$  INSTANCEMEMBER then  

  if overrideMod  $\sqcap$  {true, undefined} then throw definitionError end if  

elsif readStatus.overriddenMember = potentialConflict or  

  writeStatus.overriddenMember = potentialConflict then  

  if overrideMod  $\sqcap$  {false, undefined} then throw definitionError end if  

else if overrideMod  $\sqcap$  {none, false, undefined} then throw definitionError end if  

end if;  

newReadBindings: INSTANCEBINDING{}  $\sqcap$   

  {INSTANCEBINDING[qname: qname, content: m]  $\sqcap$  qname  $\sqcap$  readStatus.multiname};  

c.instanceReadBindings  $\sqcup$  c.instanceReadBindings  $\sqcap$  newReadBindings;  

newWriteBindings: INSTANCEBINDING{}  $\sqcap$   

  {INSTANCEBINDING[qname: qname, content: m]  $\sqcap$  qname  $\sqcap$  writeStatus.multiname};  

c.instanceWriteBindings  $\sqcup$  c.instanceWriteBindings  $\sqcap$  newWriteBindings;  

return OVERRIDESTATUSPAIR[readStatus: readStatus, writeStatus: writeStatus]  

end proc;

```

10.5.4 Instantiation

```

proc instantiateOpenInstance(oi: OPENINSTANCE, env: ENVIRONMENT): INSTANCE  

  cache: CALLABLEINSTANCE  $\sqcap$  {none}  $\sqcap$  oi.cache;  

if cache = none then  

  slots: SLOT{}  $\sqcap$  {new SLOT[id: s.id, value: s.value]  $\sqcap$  s  $\sqcap$  oi.defaultSlots};  

  dynamicProperties: DYNAMICPROPERTY{}  $\sqcap$  {fixed};  

if oi.buildPrototype then dynamicProperties  $\sqcap$  {};????  

else dynamicProperties  $\sqcap$  fixed  

end if;  

i: CALLABLEINSTANCE  $\sqcap$  new CALLABLEINSTANCE[type: oi.type, typeofString: oi.typeofString, slots: slots,  

  dynamicProperties: dynamicProperties, call: oi.call, construct: oi.construct, env: env]  

reuse: BOOLEAN;  

At the implementation's discretion, either reuse  $\sqcap$  true, or reuse  $\sqcap$  false. An implementation may make different  

choices at different times. The intent here is to allow implementations the freedom to reuse a closure object  

rather than create a new closure each time a particular OPENINSTANCE is instantiated if the implementation  

notices that the resulting closures would be behaviorally indistinguishable from each other.  

if reuse then oi.cache  $\sqcap$  i end if;  

return i  

else return new ALIASINSTANCE[original: cache, env: env]  

end if  

end proc;

```

```

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

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

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

```

10.5.5 Environmental Lookup

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

```

proc findThis(env: ENVIRONMENT, allowPrototypeThis: BOOLEAN): OBJECTOPT
  for each frame  $\sqsubseteq$  env do
    if frame  $\sqsubseteq$  PARAMETERFRAME and frame.this  $\neq$  none then
      if allowPrototypeThis or not frame.prototype then return frame.this end if
    end if
  end for each;
  return none
end proc;

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

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

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

```

10.5.6 Property Lookup

tag **propertyLookup**:

```
tuple LEXICALLOOKUP
  this: OBJECTOPT
end tuple;

LOOKUPKIND = {propertyLookup} □ LEXICALLOOKUP;

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

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

```

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

```

```

proc resolveInstanceMemberName(c: CLASS, multiname: MULTINAME, access: {read, write}, phase: PHASE):
    QUALENAMEOPT
    Start from the root class (Object) and proceed through more specific classes that are ancestors of c.
    for each s  $\sqsubseteq$  ancestors(c) do
        matchingInstanceBindings: INSTANCEBINDING{}  $\sqcup$  {b |  $\sqsubseteq$  b  $\sqsubseteq$  instanceBindingsWithAccess(s, access) such that
            b.qname  $\sqsubseteq$  multiname};
        Note that if the same INSTANCEMEMBER was found via several different bindings b, then it will appear only once in
        the set matchingMembers.
        matchingInstanceMembers: INSTANCEMEMBER{}  $\sqcup$  {b.content |  $\sqsubseteq$  b  $\sqsubseteq$  matchingInstanceBindings};
        if matchingInstanceMembers  $\neq$  {} then
            if |matchingInstanceMembers| = 1 then
                Return the qualified name of any matching binding. It doesn't matter which because they all refer to the same
                INSTANCEMEMBER, and if one is overridden by a subclass then all must be overridden in the same way
                by that subclass.
                b: INSTANCEBINDING  $\sqsubseteq$  any element of matchingInstanceBindings;
                return b.qname
            else
                This access is ambiguous because the bindings it found belong to several different members in the same class.
                throw propertyAccessError
            end if
        end if
    end for each;
    return none
end proc;

proc findInstanceMember(c: CLASSOPT, qname: QUALENAMEOPT, access: {read, write}): INSTANCEMEMBEROPT
    if qname = none then return none end if;
    s: CLASSOPT  $\sqsubseteq$  c;
    while s  $\neq$  none do
        if some b  $\sqsubseteq$  instanceBindingsWithAccess(s, access) satisfies b.qname = qname then
            return b.content
        end if;
        s  $\sqsubseteq$  s.super
    end while;
    return none
end proc;

```

10.5.7 Reading a Property

tag generic;

```

proc readProperty(container: OBJOPTIONALLIMIT [] FRAME, multiname: MULTINAME, kind: LOOKUPKIND,
phase: PHASE): OBJECTOPT
  case container of
    UNDEFINED [] NULL [] BOOLEAN [] GENERALNUMBER [] CHARACTER [] STRING [] NAMESPACE []
      COMPOUNDATTRIBUTE [] METHODCLOSURE [] INSTANCE do
        c: CLASS [] objectType(container);
        qname: QUALIFIEDNAMEOPT [] resolveInstanceMemberName(c, multiname, read, phase);
        if qname = none and container [] INSTANCE then
          return readDynamicProperty(resolveAlias(container), multiname, kind, phase)
        else return readInstanceMember(container, c, qname, phase)
        end if;
    SYSTEMFRAME [] GLOBAL [] PACKAGE [] PARAMETERFRAME [] BLOCKFRAME do
      m: STATICMEMBEROPT [] findFlatMember(container, multiname, read, phase);
      if m = none and container [] GLOBAL then
        return readDynamicProperty(container, multiname, kind, phase)
      else return readStaticMember(m, phase)
      end if;
    CLASS do
      this: OBJECT [] {inaccessible, none, generic};
      case kind of
        {propertyLookup} do this [] generic;
        LEXICALLOOKUP do this [] kind.this
      end case;
      m2: {none} [] STATICMEMBER [] QUALIFIEDNAME [] findStaticMember(container, multiname, read, phase);
      if m2 [] QUALIFIEDNAME then return readStaticMember(m2, phase) end if;
      case this of
        {none} do throw propertyAccessError;
        {inaccessible} do throw compileExpressionError;
        {generic} do ????
        OBJECT do return readInstanceMember(this, objectType(this), m2, phase)
      end case;
      PROTOTYPE do return readDynamicProperty(container, multiname, kind, phase);
      LIMITEDINSTANCE do
        superclass: CLASSOPT [] container.limit.super;
        if superclass = none then return none end if;
        qname: QUALIFIEDNAMEOPT [] resolveInstanceMemberName(superclass, multiname, read, phase);
        return readInstanceMember(container.instance, superclass, qname, phase)
      end case
    end proc;

proc readInstanceMember(this: OBJECT, c: CLASS, qname: QUALIFIEDNAMEOPT, phase: PHASE): OBJECTOPT
  m: INSTANCEMEMBEROPT [] findInstanceMember(c, qname, read);
  case m of
    {none} do return none;
    INSTANCEVARIABLE do
      if phase = compile and not m.immutable then throw compileExpressionError
      end if;
      v: OBJECTU [] findSlot(this, m).value;
      if v = uninitialised then throw uninitialisedError end if;
      return v;
    INSTANCEMETHOD do return METHODCLOSURE[this: this, method: m]
    INSTANCEGETTER do return m.call(this, m.env, phase);
    INSTANCESETTER do
      m cannot be an INSTANCESETTER because these are only represented as write-only members.
    end case
  end proc;

```

```

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

proc readDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME, kind: LOOKUPKIND, phase: PHASE):
  OBJECTOPT
  name: STRINGOPT  $\sqcup$  selectPublicName(multiname);
  if name = none then return none end if;
  if phase = compile then throw compileExpressionError end if;
  dynamicProperties: DYNAMICPROPERTY{}  $\sqcup$  {fixed}  $\sqcup$  container.dynamicProperties;
  if dynamicProperties ≠ fixed and (some dp  $\sqcup$  dynamicProperties satisfies dp.name = name) then
    return dp.value
  end if;
  if container  $\sqcup$  PROTOTYPE then
    parent: PROTOTYPEOPT  $\sqcup$  container.parent;
    if parent ≠ none then return readDynamicProperty(parent, multiname, kind, phase)
    end if
  end if;
  if kind = propertyLookup then return undefined end if;
  return none
end proc;

```

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

10.5.8 Writing a Property

```

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

```

```

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

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

```

```

proc writeDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME, createIfMissing: BOOLEAN,
  newValue: OBJECT, phase: {run}): {none, ok}
  name: STRINGOPT  $\sqcup$  selectPublicName(multiname);
  if name = none then return none end if;
  dynamicProperties: DYNAMICPROPERTY{}  $\sqcup$  {fixed}  $\sqcup$  container.dynamicProperties;
  if dynamicProperties = fixed then return none end if;
  if some dp  $\sqcup$  dynamicProperties satisfies dp.name = name then
    dp.value  $\sqcup$  newValue;
    return ok
  end if;
  if not createIfMissing then return none end if;
Before trying to create a new dynamic property, check that there is no read-only fixed property with the same name.
m: {none}  $\sqcup$  STATICMEMBER  $\sqcup$  QUALIFIEDNAME;
case container of
  PROTOTYPE do m  $\sqcup$  none;
  SIMPLEINSTANCE  $\sqcup$  CALLABLEINSTANCE do
    m  $\sqcup$  resolveInstanceMemberName(objectType(container), multiname, read, phase);
    GLOBAL do m  $\sqcup$  findFlatMember(container, multiname, read, phase)
  end case;
  if m  $\neq$  none then return none end if;
  container.dynamicProperties  $\sqcup$  dynamicProperties  $\sqcup$  {new DYNAMICPROPERTY[]|name: name, value: newValue[]};
  return ok
end proc;

proc getVariableType(v: VARIABLE, phase: PHASE): CLASS
  type: VARIABLETYPE  $\sqcup$  v.type;
  case type of
    CLASS do return type;
    {inaccessible} do
      Note that this can only happen when phase = compile because the compilation phase ensures that all types are
      valid, so invalid types will not occur during the run phase.
      throw compileExpressionError;
    ()  $\sqcup$  CLASS do
      Note that phase = compile because all futures are resolved by the end of the compilation phase.
      v.type  $\sqcup$  inaccessible;
      newType: CLASS  $\sqcup$  type();
      v.type  $\sqcup$  newType;
      return newType
    end case
end proc;

proc writeVariable(v: VARIABLE, newValue: OBJECT, phase: {run})
  type: CLASS  $\sqcup$  getVariableType(v, phase);
  if v.value = inaccessible or (v.immutable and v.value  $\neq$  uninitialized) then
    throw propertyAccessError
  end if;
  coercedValue: OBJECT  $\sqcup$  type.implicitCoerce(newValue);
  v.value  $\sqcup$  coercedValue
end proc;

```

10.5.9 Deleting a Property

```

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

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

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

```

```

proc deleteDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME): BOOLEANOPT
  name: STRINGOPT  $\sqcap$  selectPublicName(multiname);
  if name = none then return none end if;
  dynamicProperties: DYNAMICPROPERTY{}  $\sqcap$  {fixed}  $\sqcap$  container.dynamicProperties;
  if dynamicProperties = fixed then return none end if;
  if some dp  $\sqcap$  dynamicProperties satisfies dp.name = name then
    container.dynamicProperties  $\sqcap$  dynamicProperties - {dp};
    return true
  else return none
  end if
end proc;

```

10.6 Invocation

```

proc badConstruct(args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
  throw propertyAccessError
end proc;

```

11 Evaluation

11.1 Phases of Evaluation

- Parse using the grammar. If the parse fails, throw a syntax error.
- Call **Validate** on the goal nonterminal, which will recursively call **Validate** on some intermediate nonterminals. This checks that the program is well-formed, ensuring for instance that **break** and **continue** labels exist, compile-time constant expressions really are compile-time constant expressions, etc. If the check fails, **Validate** will throw an exception.
- [Call Setup on the goal nonterminal, which will recursively call Setup on some intermediate nonterminals.](#)
- Call **Eval** on the goal nonterminal.

11.2 Constant Expressions

12 Expressions

Some expression grammar productions in this chapter are parameterised (see section 5.14.4) by the grammar argument \sqcap \sqcap {allowIn, noIn}

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

12.1 Identifiers

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

Syntax

```
Identifier □
  Identifier
  | get
  | set
  | exclude
  | include
  | named
```

Semantics

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

12.2 Qualified Identifiers

Syntax

```
Qualifier □
  Identifier
  | public
  | private
```

```
SimpleQualifiedIdentifier □
  Identifier
  | Qualifier :: Identifier
```

```
ExpressionQualifiedIdentifier □ ParenExpression :: Identifier
```

```
QualifiedIdentifier □
  SimpleQualifiedIdentifier
  | ExpressionQualifiedIdentifier
```

Validation

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

```
Multiname[SimpleQualifiedIdentifier]: MULTINAME;
```

```

proc Validate[SimpleQualifiedIdentifier] (ctxt: CONTEXT, env: ENVIRONMENT)
  [SimpleQualifiedIdentifier  $\sqsubseteq$  Identifier] do
    multiname: MULTINAME  $\sqsubseteq$  {QUALIFIEDNAME[namespace: ns, id: Name[Identifier]]}
     $\sqcup$  ns  $\sqcup$  ctxt.openNamespaces;
    Multiname[SimpleQualifiedIdentifier]  $\sqsubseteq$  multiname;
  [SimpleQualifiedIdentifier  $\sqsubseteq$  Qualifier :: Identifier] do
    q: NAMESPACE  $\sqsubseteq$  Validate[Qualifier](ctxt, env);
    Multiname[SimpleQualifiedIdentifier]  $\sqsubseteq$  {QUALIFIEDNAME[namespace: q, id: Name[Identifier]]}
  end proc;

Multiname[ExpressionQualifiedIdentifier]: MULTINAME;

proc Validate[ExpressionQualifiedIdentifier  $\sqsubseteq$  ParenExpression :: Identifier] (ctxt: CONTEXT, env: ENVIRONMENT)
  Validate[ParenExpression](ctxt, env);
  Setup[ParenExpression]();
  r: OBJORREF  $\sqsubseteq$  Eval[ParenExpression](env, compile);
  q: OBJECT  $\sqsubseteq$  readReference(r, compile);
  if q  $\sqsubseteq$  NAMESPACE then throw badValueError end if;
  Multiname[ExpressionQualifiedIdentifier]  $\sqsubseteq$  {QUALIFIEDNAME[namespace: q, id: Name[Identifier]]}
end proc;

Multiname[QualifiedIdentifier]: MULTINAME;

proc Validate[QualifiedIdentifier] (ctxt: CONTEXT, env: ENVIRONMENT)
  [QualifiedIdentifier  $\sqsubseteq$  SimpleQualifiedIdentifier] do
    Validate[SimpleQualifiedIdentifier](ctxt, env);
    Multiname[QualifiedIdentifier]  $\sqsubseteq$  Multiname[SimpleQualifiedIdentifier];
  [QualifiedIdentifier  $\sqsubseteq$  ExpressionQualifiedIdentifier] do
    Validate[ExpressionQualifiedIdentifier](ctxt, env);
    Multiname[QualifiedIdentifier]  $\sqsubseteq$  Multiname[ExpressionQualifiedIdentifier]
  end proc;

```

Setup

```

proc Setup[SimpleQualifiedIdentifier] ()
  [SimpleQualifiedIdentifier  $\sqsubseteq$  Identifier] do nothing;
  [SimpleQualifiedIdentifier  $\sqsubseteq$  Qualifier :: Identifier] do nothing
end proc;

proc Setup[ExpressionQualifiedIdentifier  $\sqsubseteq$  ParenExpression :: Identifier] ()
end proc;

```

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

12.3 Primary Expressions

Syntax

```

PrimaryExpression ::= 
  null
  | true
  | false
  | public
  | Number
  | String
  | this
  | RegularExpression
  | ParenListExpression
  | ArrayLiteral
  | ObjectLiteral
  | FunctionExpression

ParenExpression ::= ( AssignmentExpressionallowIn )

ParenListExpression ::= 
  ParenExpression
  | ( ListExpressionallowIn, AssignmentExpressionallowIn )

```

Validation

```

proc Validate[PrimaryExpression] (ctxt: CONTEXT, env: ENVIRONMENT)
  [PrimaryExpression ::= null] do nothing;
  [PrimaryExpression ::= true] do nothing;
  [PrimaryExpression ::= false] do nothing;
  [PrimaryExpression ::= public] do nothing;
  [PrimaryExpression ::= Number] do nothing;
  [PrimaryExpression ::= String] do nothing;
  [PrimaryExpression ::= this] do
    if findThis(env, true) = none then throw syntaxError end if;
  [PrimaryExpression ::= RegularExpression] do nothing;
  [PrimaryExpression ::= ParenListExpression] do
    Validate[ParenListExpression](ctxt, env);
  [PrimaryExpression ::= ArrayLiteral] do Validate[ArrayLiteral](ctxt, env);
  [PrimaryExpression ::= ObjectLiteral] do Validate[ObjectLiteral](ctxt, env);
  [PrimaryExpression ::= FunctionExpression] do Validate[FunctionExpression](ctxt, env)
end proc;

```

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

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

Setup

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

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

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

Evaluation

```

proc Eval[PrimaryExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [PrimaryExpression  $\sqsubseteq$  null] do return null;
  [PrimaryExpression  $\sqsubseteq$  true] do return true;
  [PrimaryExpression  $\sqsubseteq$  false] do return false;
  [PrimaryExpression  $\sqsubseteq$  public] do return publicNamespace;
  [PrimaryExpression  $\sqsubseteq$  Number] do return Value[Number];
  [PrimaryExpression  $\sqsubseteq$  String] do return Value[String];
  [PrimaryExpression  $\sqsubseteq$  this] do
    this: OBJECTOPT  $\sqsubseteq$  findThis(env, true);
    Note that Validate ensured that this cannot be none at this point.
    if this = inaccessible then throw compileExpressionError end if;
    return this;
  [PrimaryExpression  $\sqsubseteq$  RegularExpression] do ????
  [PrimaryExpression  $\sqsubseteq$  ParenListExpression] do
    return Eval[ParenListExpression](env, phase);
  [PrimaryExpression  $\sqsubseteq$  ArrayLiteral] do return Eval[ArrayLiteral](env, phase);
  [PrimaryExpression  $\sqsubseteq$  ObjectLiteral] do return Eval[ObjectLiteral](env, phase);
  [PrimaryExpression  $\sqsubseteq$  FunctionExpression] do
    return Eval[FunctionExpression](env, phase)
end proc;

proc Eval[ParenExpression  $\sqsubseteq$  (AssignmentExpressionallowIn)] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  return Eval[AssignmentExpressionallowIn](env, phase)
end proc;

proc Eval[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ParenListExpression  $\sqsubseteq$  ParenExpression] do return Eval[ParenExpression](env, phase);
  [ParenListExpression  $\sqsubseteq$  (ListExpressionallowIn, AssignmentExpressionallowIn)] do
    ra: OBJORREF  $\sqsubseteq$  Eval[ListExpressionallowIn](env, phase);
    readReference(ra, phase);
    rb: OBJORREF  $\sqsubseteq$  Eval[AssignmentExpressionallowIn](env, phase);
    return readReference(rb, phase)
end proc;

proc EvalAsList[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
  [ParenListExpression  $\sqsubseteq$  ParenExpression] do
    r: OBJORREF  $\sqsubseteq$  Eval[ParenExpression](env, phase);
    elt: OBJECT  $\sqsubseteq$  readReference(r, phase);
    return [elt];
  [ParenListExpression  $\sqsubseteq$  (ListExpressionallowIn, AssignmentExpressionallowIn)] do
    elts: OBJECT[]  $\sqsubseteq$  EvalAsList[ListExpressionallowIn](env, phase);
    r: OBJORREF  $\sqsubseteq$  Eval[AssignmentExpressionallowIn](env, phase);
    elt: OBJECT  $\sqsubseteq$  readReference(r, phase);
    return elts  $\oplus$  [elt]
end proc;
```

12.4 Function Expressions

Syntax

```
FunctionExpression □
  function FunctionCommon
  | function Identifier FunctionCommon
```

Validation

```
F[FunctionExpression]: OPENINSTANCE;

proc Validate[FunctionExpression] (ctx: CONTEXT, env: ENVIRONMENT)
  [FunctionExpression □ function FunctionCommon] do
    unchecked: BOOLEAN □ not ctx.strict and Untyped[FunctionCommon];
    Unchecked[FunctionCommon] □ unchecked;
    this: {none, inaccessible} □ unchecked? inaccessible : none;
    F[FunctionExpression] □ ValidateStaticFunction[FunctionCommon](ctx, env, this, unchecked);
    [FunctionExpression □ function Identifier FunctionCommon] do ????
  end proc;
```

Setup

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

Evaluation

```
proc Eval[FunctionExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [FunctionExpression □ function FunctionCommon] do
    if phase = compile then throw compileExpressionError end if;
    return instantiateOpenInstance(F[FunctionExpression], env);
  [FunctionExpression □ function Identifier FunctionCommon] do
    if phase = compile then throw compileExpressionError end if;
    return instantiateOpenInstance(F[FunctionExpression], env)
  end proc;
```

12.5 Object Literals

Syntax

```
ObjectLiteral □
  { }
  | { FieldList }

FieldList □
  LiteralField
  | FieldList , LiteralField

LiteralField □ FieldName : AssignmentExpressionallowIn
```

```

FieldName □
  Identifier
  | String
  | Number

```

Validation

```

proc Validate[ObjectLiteral] (ctx: CONTEXT, env: ENVIRONMENT)
  [ObjectLiteral □ { } ] do nothing;
  [ObjectLiteral □ { FieldList } ] do Validate[FieldList](ctx, env)
end proc;

proc Validate[FieldList] (ctx: CONTEXT, env: ENVIRONMENT): STRING {}
  [FieldList □ LiteralField] do return Validate[LiteralField](ctx, env);
  [FieldList0 □ FieldList1, LiteralField] do
    names1: STRING {} □ Validate[FieldList1](ctx, env);
    names2: STRING {} □ Validate[LiteralField](ctx, env);
    if names1 □ names2 ≠ {} then throw syntaxError end if;
    return names1 □ names2
  end proc;

proc Validate[LiteralField □ FieldName : AssignmentExpressionallowIn] (ctx: CONTEXT, env: ENVIRONMENT): STRING {}
  names: STRING {} □ Validate[FieldName](ctx, env);
  Validate[AssignmentExpressionallowIn](ctx, env);
  return names
end proc;

proc Validate[FieldName] (ctx: CONTEXT, env: ENVIRONMENT): STRING {}
  [FieldName □ Identifier] do return {Name[Identifier]};
  [FieldName □ String] do return {Value[String]};
  [FieldName □ Number] do return {toString(Value[Number], compile)}
end proc;

```

Setup

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

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

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

```

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

```

Evaluation

```

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

```

```

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

proc Eval[FieldList] (env: ENVIRONMENT, phase: PHASE): DYNAMICPROPERTY{}
  [FieldList ⊑ LiteralField] do
    na: NAMEDARGUMENT ⊑ Eval[LiteralField](env, phase);
    return {new DYNAMICPROPERTY[] name: na.name, value: na.value[]};
  [FieldList0 ⊑ FieldList1 , LiteralField] do
    properties: DYNAMICPROPERTY{} ⊑ Eval[FieldList1](env, phase);
    na: NAMEDARGUMENT ⊑ Eval[LiteralField](env, phase);
    if some p ⊑ properties satisfies p.name = na.name then
      throw argumentMismatchError
    end if;
    return properties ⊑ {new DYNAMICPROPERTY[] name: na.name, value: na.value[]}
  end proc;

  proc Eval[LiteralField ⊑ FieldName : AssignmentExpressionallowIn]
    (env: ENVIRONMENT, phase: PHASE): NAMEDARGUMENT
    name: STRING ⊑ Eval[FieldName](env, phase);
    r: OBJORREF ⊑ Eval[AssignmentExpressionallowIn](env, phase);
    value: OBJECT ⊑ readReference(r, phase);
    return NAMEDARGUMENT[] name: name, value: value[]
  end proc;

  proc Eval[FieldName] (env: ENVIRONMENT, phase: PHASE): STRING
    [FieldName ⊑ Identifier] do return Name[Identifier];
    [FieldName ⊑ String] do return Value[String];
    [FieldName ⊑ Number] do return toString(Value[Number], compile)
  end proc;

```

12.6 Array Literals

Syntax

ArrayLiteral ⊑ [*ElementList*]

ElementList ⊑
LiteralElement
 | *ElementList* , *LiteralElement*

LiteralElement ⊑
 «empty»
 | *AssignmentExpression*^{allowIn}

Validation

```

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

```

Setup

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

Evaluation

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

12.7 Super Expressions

Syntax

SuperExpression []
 super
 | super *ParenExpression*

Validation

```
proc Validate[SuperExpression] (ctx: CONTEXT, env: ENVIRONMENT)  
    [SuperExpression [] super] do  
        if getEnclosingClass(env) = none or findThis(env, false) = none then  
            throw syntaxError  
        end if,  
        [SuperExpression [] super ParenExpression] do  
            if getEnclosingClass(env) = none then throw syntaxError end if,  
            Validate[ParenExpression](ctx, env)  
    end proc;
```

Setup

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

Evaluation

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

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

```
proc readLimitedReference(r: OBJORREF, limit: CLASS, phase: PHASE): OBJOPTIONALLIMIT
  o: OBJECT  $\sqcup$  readReference(r, phase);
  if o = null then return null end if;
  if o  $\sqcup$  INSTANCE or not hasType(o, limit) then throw badValueError end if;
  return LIMITEDINSTANCE[instance: o, limit: limit]
end proc;
```

12.8 Postfix Expressions

Syntax

```
PostfixExpression  $\sqcup$ 
  AttributeExpression
  | FullPostfixExpression
  | ShortNewExpression

AttributeExpression  $\sqcup$ 
  SimpleQualifiedIdentifier
  | AttributeExpression MemberOperator
  | AttributeExpression Arguments

FullPostfixExpression  $\sqcup$ 
  PrimaryExpression
  | ExpressionQualifiedIdentifier
  | FullNewExpression
  | FullPostfixExpression MemberOperator
  | SuperExpression MemberOperator
  | FullPostfixExpression Arguments
  | PostfixExpression [no line break] ++
  | PostfixExpression [no line break] --

FullNewExpression  $\sqcup$  new FullNewSubexpression Arguments

FullNewSubexpression  $\sqcup$ 
  PrimaryExpression
  | QualifiedIdentifier
  | FullNewExpression
  | FullNewSubexpression MemberOperator
  | SuperExpression MemberOperator

ShortNewExpression  $\sqcup$  new ShortNewSubexpression

ShortNewSubexpression  $\sqcup$ 
  FullNewSubexpression
  | ShortNewExpression
```

Validation

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

Strict[*AttributeExpression*]: BOOLEAN;

```
proc Validate[AttributeExpression] (ctx: CONTEXT, env: ENVIRONMENT)
  [AttributeExpression  $\sqcup$  SimpleQualifiedIdentifier] do
    Validate[SimpleQualifiedIdentifier](ctx, env);
    Strict[AttributeExpression]  $\sqcup$  ctx.strict;
```

```

[AttributeExpression0 □ AttributeExpression1 MemberOperator] do
  Validate[AttributeExpression1](ctxt, env);
  Validate[MemberOperator](ctxt, env);
[AttributeExpression0 □ AttributeExpression1 Arguments] do
  Validate[AttributeExpression1](ctxt, env);
  Validate[Arguments](ctxt, env)
end proc;

Strict[FullPostfixExpression]: BOOLEAN;

proc Validate[FullPostfixExpression] (ctxt: CONTEXT, env: ENVIRONMENT)
  [FullPostfixExpression □ PrimaryExpression] do
    Validate[PrimaryExpression](ctxt, env);
  [FullPostfixExpression □ ExpressionQualifiedIdentifier] do
    Validate[ExpressionQualifiedIdentifier](ctxt, env);
    Strict[FullPostfixExpression] □ ctxt.strict;
  [FullPostfixExpression □ FullNewExpression] do
    Validate[FullNewExpression](ctxt, env);
  [FullPostfixExpression0 □ FullPostfixExpression1 MemberOperator] do
    Validate[FullPostfixExpression1](ctxt, env);
    Validate[MemberOperator](ctxt, env);
  [FullPostfixExpression □ SuperExpression MemberOperator] do
    Validate[SuperExpression](ctxt, env);
    Validate[MemberOperator](ctxt, env);
  [FullPostfixExpression0 □ FullPostfixExpression1 Arguments] do
    Validate[FullPostfixExpression1](ctxt, env);
    Validate[Arguments](ctxt, env);
  [FullPostfixExpression □ PostfixExpression [no line break] ++] do
    Validate[PostfixExpression](ctxt, env);
  [FullPostfixExpression □ PostfixExpression [no line break] --] do
    Validate[PostfixExpression](ctxt, env)
end proc;

```

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

```

Strict[FullNewSubexpression]: BOOLEAN;

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

```

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

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

Setup

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

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

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

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

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

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

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

Evaluation

```

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

proc Eval[AttributeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [AttributeExpression □ SimpleQualifiedIdentifier] do
    return LEXICALREFERENCE[env: env, variableMultiname: Multiname[SimpleQualifiedIdentifier],
      strict: Strict[AttributeExpression]];
  [AttributeExpression_0 □ AttributeExpression_1 MemberOperator] do
    r: OBJORREF □ Eval[AttributeExpression]_1(env, phase);
    a: OBJECT □ readReference(r, phase);
    return Eval[MemberOperator](env, a, phase);
  [AttributeExpression_0 □ AttributeExpression_1 Arguments] do
    r: OBJORREF □ Eval[AttributeExpression]_1(env, phase);
    f: OBJECT □ readReference(r, phase);
    base: OBJECT □ referenceBase(r);
    args: ARGUMENTLIST □ Eval[Arguments](env, phase);
    return call(base, f, args, phase)
end proc;

proc Eval[FullPostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [FullPostfixExpression □ PrimaryExpression] do
    return Eval[PrimaryExpression](env, phase);

```

```

[FullPostfixExpression □ ExpressionQualifiedIdentifier] do
  return LEXICALREFERENCE[env: env, variableMultiname: Multiname[ExpressionQualifiedIdentifier],
    strict: Strict[FullPostfixExpression]()]
[FullPostfixExpression □ FullNewExpression] do
  return Eval[FullNewExpression](env, phase);
[FullPostfixExpression_0 □ FullPostfixExpression_1 MemberOperator] do
  r: OBJORREF □ Eval[FullPostfixExpression_1](env, phase);
  a: OBJECT □ readReference(r, phase);
  return Eval[MemberOperator](env, a, phase);
[FullPostfixExpression □ SuperExpression MemberOperator] do
  a: OBJOPTIONALLIMIT □ Eval[SuperExpression](env, phase);
  return Eval[MemberOperator](env, a, phase);
[FullPostfixExpression_0 □ FullPostfixExpression_1 Arguments] do
  r: OBJORREF □ Eval[FullPostfixExpression_1](env, phase);
  f: OBJECT □ readReference(r, phase);
  base: OBJECT □ referenceBase(r);
  args: ARGUMENTLIST □ Eval[Arguments](env, phase);
  return call(base, f, args, phase);
[FullPostfixExpression □ PostfixExpression [no line break] ++] do
  if phase = compile then throw compileExpressionError end if;
  r: OBJORREF □ Eval[PostfixExpression](env, phase);
  a: OBJECT □ readReference(r, phase);
  b: OBJECT □ plus(a, phase);
  c: OBJECT □ add(b, 1.0f64, phase);
  writeReference(r, c, phase);
  return b;
[FullPostfixExpression □ PostfixExpression [no line break] --] do
  if phase = compile then throw compileExpressionError end if;
  r: OBJORREF □ Eval[PostfixExpression](env, phase);
  a: OBJECT □ readReference(r, phase);
  b: OBJECT □ plus(a, phase);
  c: OBJECT □ subtract(b, 1.0f64, phase);
  writeReference(r, c, phase);
  return b;
end proc;

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

proc Eval[FullNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [FullNewSubexpression □ PrimaryExpression] do
    return Eval[PrimaryExpression](env, phase);
  [FullNewSubexpression □ QualifiedIdentifier] do
    return LEXICALREFERENCE[env: env, variableMultiname: Multiname[QualifiedIdentifier],
      strict: Strict[FullNewSubexpression]()]
  [FullNewSubexpression □ FullNewExpression] do
    return Eval[FullNewExpression](env, phase);

```

```

[FullNewSubexpression0 ⊑ FullNewSubexpression1 MemberOperator] do
  r: OBJORREF ⊑ Eval[FullNewSubexpression1](env, phase);
  a: OBJECT ⊑ readReference(r, phase);
  return Eval[MemberOperator](env, a, phase);

[FullNewSubexpression ⊑ SuperExpression MemberOperator] do
  a: OBJOPTIONALLIMIT ⊑ Eval[SuperExpression](env, phase);
  return Eval[MemberOperator](env, a, phase)

end proc;

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

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

```

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

```

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

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

```

```

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

```

12.9 Member Operators

Syntax

MemberOperator □
 | . QualifiedIdentifier
 | Brackets

Brackets □
 | []
 | [*ListExpression*^{allowIn}]
 | [*NamedArgumentList*]

Arguments □
 | ParenExpressions
 | (*NamedArgumentList*)

ParenExpressions □
 | ()
 | ParenListExpression

NamedArgumentList □
 | LiteralField
 | *ListExpression*^{allowIn}, LiteralField
 | *NamedArgumentList*, LiteralField

Validation

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

```

proc Validate[Brackets] (ctxt: CONTEXT, env: ENVIRONMENT)
  Brackets □ [ ] do nothing;
  Brackets □ [ ListExpressionallowIn ] do Validate[ListExpressionallowIn](ctxt, env);
  Brackets □ [ NamedArgumentList ] do Validate[NamedArgumentList](ctxt, env)
end proc;

```

```

proc Validate[Arguments] (ctx: CONTEXT, env: ENVIRONMENT)
  [Arguments □ ParenExpressions] do Validate[ParenExpressions](ctx, env);
  [Arguments □ ( NamedArgumentList )] do Validate[NamedArgumentList](ctx, env)
end proc;

```

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

```

proc Validate[NamedArgumentList] (ctx: CONTEXT, env: ENVIRONMENT): STRING{}
  [NamedArgumentList □ LiteralField] do return Validate[LiteralField](ctx, env);
  [NamedArgumentList □ ListExpressionallowIn, LiteralField] do
    Validate[ListExpressionallowIn](ctx, env);
  return Validate[LiteralField](ctx, env);
  [NamedArgumentList0 □ NamedArgumentList1, LiteralField] do
    names1: STRING{} □ Validate[NamedArgumentList1](ctx, env);
    names2: STRING{} □ Validate[LiteralField](ctx, env);
    if names1 □ names2 ≠ {} then throw syntaxError end if;
    return names1 □ names2
end proc;

```

Setup

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

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

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

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

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

Evaluation

```

proc Eval[MemberOperator] (env: ENVIRONMENT, base: OBJOPTIONALLIMIT, phase: PHASE): OBJORREF
  [MemberOperator □ . QualifiedIdentifier] do
    return DOTREFERENCE[base: base, propertyMultiname: Multiname[QualifiedIdentifier]()]
  [MemberOperator □ Brackets] do
    args: ARGUMENTLIST □ Eval[Brackets](env, phase);
    return BRACKETREFERENCE[base: base, args: args()]
end proc;

```

```

proc Eval[Brackets] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
  [Brackets □ [ ]] do return ARGUMENTLIST[positional: [], named: {}]
  [Brackets □ [ ListExpressionallowIn ]] do
    positional: OBJECT[] □ EvalAsList[ListExpressionallowIn](env, phase);
    return ARGUMENTLIST[positional: positional, named: {}]
  [Brackets □ [ NamedArgumentList ]] do return Eval[NamedArgumentList](env, phase)
end proc;

```

```

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

```

```

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

proc Eval[NamedArgumentList] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
  [NamedArgumentList □ LiteralField] do
    na: NAMEDARGUMENT □ Eval[LiteralField](env, phase);
    return ARGUMENTLIST[positional: [], named: {na}]
  [NamedArgumentList □ ListExpressionallowIn, LiteralField] do
    positional: OBJECT[] □ EvalAsList[ListExpressionallowIn](env, phase);
    na: NAMEDARGUMENT □ Eval[LiteralField](env, phase);
    return ARGUMENTLIST[positional: positional, named: {na}]
  [NamedArgumentList0 □ NamedArgumentList1, LiteralField] do
    args: ARGUMENTLIST □ Eval[NamedArgumentList1](env, phase);
    na: NAMEDARGUMENT □ Eval[LiteralField](env, phase);
    if some na2 □ args.named satisfies na2.name = na.name then
      throw argumentMismatchError
    end if;
    return ARGUMENTLIST[positional: args.positional, named: args.named □ {na}]
end proc;

```

12.10 Unary Operators

Syntax

```

UnaryExpression □
  PostfixExpression
  | delete PostfixExpression
  | void UnaryExpression
  | typeof UnaryExpression
  | ++ PostfixExpression
  | -- PostfixExpression
  | + UnaryExpression
  | - UnaryExpression
  | - NegatedMinLong
  | ~ UnaryExpression
  | ! UnaryExpression

```

Validation

```

Strict[UnaryExpression]: BOOLEAN;

proc Validate[UnaryExpression] (ctxt: CONTEXT, env: ENVIRONMENT)
  [UnaryExpression □ PostfixExpression] do Validate[PostfixExpression](ctxt, env);
  [UnaryExpression □ delete PostfixExpression] do
    Validate[PostfixExpression](ctxt, env);
    Strict[UnaryExpression] □ ctxt.strict;
  [UnaryExpression0 □ void UnaryExpression1] do Validate[UnaryExpression1](ctxt, env);
  [UnaryExpression0 □ typeof UnaryExpression1] do
    Validate[UnaryExpression1](ctxt, env);

```

```
[UnaryExpression | ++ PostfixExpression] do Validate[PostfixExpression](ctxt, env);
[UnaryExpression | -- PostfixExpression] do Validate[PostfixExpression](ctxt, env);
[UnaryExpression0 | + UnaryExpression1] do Validate[UnaryExpression1](ctxt, env);
[UnaryExpression0 | - UnaryExpression1] do Validate[UnaryExpression1](ctxt, env);
[UnaryExpression | - NegatedMinLong] do nothing;
[UnaryExpression0 | ~ UnaryExpression1] do Validate[UnaryExpression1](ctxt, env);
[UnaryExpression0 | ! UnaryExpression1] do Validate[UnaryExpression1](ctxt, env)
end proc;
```

Setup

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

Evaluation

```
proc Eval[UnaryExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [UnaryExpression | PostfixExpression] do return Eval[PostfixExpression](env, phase);
  [UnaryExpression | delete PostfixExpression] do
    if phase = compile then throw compileExpressionError end if;
    r: OBJORREF | Eval[PostfixExpression](env, phase);
    return deleteReference(r, Strict[UnaryExpression], phase);
  [UnaryExpression0 | void UnaryExpression1] do
    r: OBJORREF | Eval[UnaryExpression1](env, phase);
    readReference(r, phase);
    return undefined;
  [UnaryExpression0 | typeof UnaryExpression1] do
    r: OBJORREF | Eval[UnaryExpression1](env, phase);
    a: OBJECT | readReference(r, phase);
    case a of
      UNDEFINED do return "undefined";
      NULL | PROTOTYPE | PACKAGE | GLOBAL do return "object";
      BOOLEAN do return "boolean";
      LONG do return "long";
      ULONG do return "ulong";
      FLOAT32 do return "float";
      FLOAT64 do return "number";
      CHARACTER do return "character";
      STRING do return "string";
      NAMESPACE do return "namespace";
      COMPOUNDATTRIBUTE do return "attribute";
      CLASS | METHODCLOSURE do return "function";
      INSTANCE do return resolveAlias(a).typeofString
    end case;
  [UnaryExpression | ++ PostfixExpression] do
    if phase = compile then throw compileExpressionError end if;
    r: OBJORREF | Eval[PostfixExpression](env, phase);
    a: OBJECT | readReference(r, phase);
    b: OBJECT | plus(a, phase);
    c: OBJECT | add(b, 1.0f64, phase);
    writeReference(r, c, phase);
    return c;
```

```

[UnaryExpression | -- PostfixExpression] do
  if phase = compile then throw compileExpressionError end if;
  r: OBJORREF | Eval[PostfixExpression](env, phase);
  a: OBJECT | readReference(r, phase);
  b: OBJECT | plus(a, phase);
  c: OBJECT | subtract(b, 1.0f64, phase);
  writeReference(r, c, phase);
  return c;

[UnaryExpression0 | + UnaryExpression1] do
  r: OBJORREF | Eval[UnaryExpression1](env, phase);
  a: OBJECT | readReference(r, phase);
  return plus(a, phase);

[UnaryExpression0 | - UnaryExpression1] do
  r: OBJORREF | Eval[UnaryExpression1](env, phase);
  a: OBJECT | readReference(r, phase);
  return minus(a, phase);

[UnaryExpression | - NegatedMinLong] do return LONG[value: -263]

[UnaryExpression0 | ~ UnaryExpression1] do
  r: OBJORREF | Eval[UnaryExpression1](env, phase);
  a: OBJECT | readReference(r, phase);
  return bitNot(a, phase);

[UnaryExpression0 | ! UnaryExpression1] do
  r: OBJORREF | Eval[UnaryExpression1](env, phase);
  a: OBJECT | readReference(r, phase);
  return logicalNot(a, phase)

end proc;

```

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

```

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

proc minus(a: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER | toGeneralNumber(a, phase);
  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  $\sqsubseteq$  toGeneralNumber(a, phase);
  case x of
    LONG do i:  $\{-2^{63} \dots 2^{63} - 1\}$   $\sqsubseteq$  x.value; return LONG[value: bitwiseXor(i, -1)];
    ULONG do
      i:  $\{0 \dots 2^{64} - 1\}$   $\sqsubseteq$  x.value;
      return ULONG[value: bitwiseXor(i, 0xFFFFFFFFFFFFFF)];
    FLOAT32  $\sqsubseteq$  FLOAT64 do
      i:  $\{-2^{31} \dots 2^{31} - 1\}$   $\sqsubseteq$  signedWrap32(truncateToInteger(x));
      return realToFloat64(bitwiseXor(i, -1));
  end case
end proc;

```

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

```

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

```

12.11 Multiplicative Operators

Syntax

```

MultiplicativeExpression  $\sqsubseteq$ 
  UnaryExpression
  | MultiplicativeExpression * UnaryExpression
  | MultiplicativeExpression / UnaryExpression
  | MultiplicativeExpression % UnaryExpression

```

Validation

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

Setup

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

Evaluation

```

proc Eval[MultiplicativeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [MultiplicativeExpression  $\sqsubseteq$  UnaryExpression] do
    return Eval[UnaryExpression](env, phase);
  [MultiplicativeExpression0  $\sqsubseteq$  MultiplicativeExpression1 * UnaryExpression] do
    ra: OBJORREF  $\sqsubseteq$  Eval[MultiplicativeExpression1](env, phase);
    a: OBJECT  $\sqsubseteq$  readReference(ra, phase);
    rb: OBJORREF  $\sqsubseteq$  Eval[UnaryExpression](env, phase);
    b: OBJECT  $\sqsubseteq$  readReference(rb, phase);
    return multiply(a, b, phase);
  [MultiplicativeExpression0  $\sqsubseteq$  MultiplicativeExpression1 / UnaryExpression] do
    ra: OBJORREF  $\sqsubseteq$  Eval[MultiplicativeExpression1](env, phase);
    a: OBJECT  $\sqsubseteq$  readReference(ra, phase);
    rb: OBJORREF  $\sqsubseteq$  Eval[UnaryExpression](env, phase);
    b: OBJECT  $\sqsubseteq$  readReference(rb, phase);
    return divide(a, b, phase);

```

```

[MultiplicativeExpression0  $\square$  MultiplicativeExpression1  $\% \text{ UnaryExpression}$ ] do
  ra: OBJORREF  $\square$  Eval[MultiplicativeExpression1](env, phase);
  a: OBJECT  $\square$  readReference(ra, phase);
  rb: OBJORREF  $\square$  Eval[UnaryExpression](env, phase);
  b: OBJECT  $\square$  readReference(rb, phase);
  return remainder(a, b, phase)
end proc;

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

```

12.12 Additive Operators

Syntax

```
AdditiveExpression ::=  
    MultiplicativeExpression  
  | AdditiveExpression + MultiplicativeExpression  
  | AdditiveExpression - MultiplicativeExpression
```

Validation

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

Setup

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

Evaluation

```
proc Eval[AdditiveExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF  
  [AdditiveExpression ∘ MultiplicativeExpression] do  
    return Eval[MultiplicativeExpression](env, phase);  
  [AdditiveExpression_0 ∘ AdditiveExpression_1 + MultiplicativeExpression] do  
    ra: OBJORREF ∘ Eval[AdditiveExpression]_1(env, phase);  
    a: OBJECT ∘ readReference(ra, phase);  
    rb: OBJORREF ∘ Eval[MultiplicativeExpression](env, phase);  
    b: OBJECT ∘ readReference(rb, phase);  
    return add(a, b, phase);  
  [AdditiveExpression_0 ∘ AdditiveExpression_1 - MultiplicativeExpression] do  
    ra: OBJORREF ∘ Eval[AdditiveExpression]_1(env, phase);  
    a: OBJECT ∘ readReference(ra, phase);  
    rb: OBJORREF ∘ Eval[MultiplicativeExpression](env, phase);  
    b: OBJECT ∘ readReference(rb, phase);  
    return subtract(a, b, phase)  
end proc;
```

```

proc add(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  ap: PRIMITIVEOBJECT  $\sqsubseteq$  toPrimitive(a, null, phase);
  bp: PRIMITIVEOBJECT  $\sqsubseteq$  toPrimitive(b, null, phase);
  if ap  $\sqsubseteq$  CHARACTER  $\sqsubseteq$  STRING or bp  $\sqsubseteq$  CHARACTER  $\sqsubseteq$  STRING then
    return toString(ap, phase)  $\oplus$  toString(bp, phase)
  end if;
  x: GENERALNUMBER  $\sqsubseteq$  toGeneralNumber(ap, phase);
  y: GENERALNUMBER  $\sqsubseteq$  toGeneralNumber(bp, phase);
  if x  $\sqsubseteq$  LONG  $\sqsubseteq$  ULONG or y  $\sqsubseteq$  LONG  $\sqsubseteq$  ULONG then
    i: INTEGEROPT  $\sqsubseteq$  checkInteger(x);
    j: INTEGEROPT  $\sqsubseteq$  checkInteger(y);
    if i  $\neq$  none and j  $\neq$  none then
      k: INTEGER  $\sqsubseteq$  i + j;
      if x  $\sqsubseteq$  ULONG or y  $\sqsubseteq$  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  $\sqsubseteq$  toGeneralNumber(a, phase);
  y: GENERALNUMBER  $\sqsubseteq$  toGeneralNumber(b, phase);
  if x  $\sqsubseteq$  LONG  $\sqsubseteq$  ULONG or y  $\sqsubseteq$  LONG  $\sqsubseteq$  ULONG then
    i: INTEGEROPT  $\sqsubseteq$  checkInteger(x);
    j: INTEGEROPT  $\sqsubseteq$  checkInteger(y);
    if i  $\neq$  none and j  $\neq$  none then
      k: INTEGER  $\sqsubseteq$  i - j;
      if x  $\sqsubseteq$  ULONG or y  $\sqsubseteq$  ULONG then return integerToULong(k)
      else return integerToLong(k)
    end if
  end if
  end if;
  return float64Subtract(toFloat64(x), toFloat64(y))
end proc;

```

12.13 Bitwise Shift Operators

Syntax

ShiftExpression \sqsubseteq
 AdditiveExpression
 | *ShiftExpression* \ll *AdditiveExpression*
 | *ShiftExpression* \gg *AdditiveExpression*
 | *ShiftExpression* \ggg *AdditiveExpression*

Validation

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

Setup

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

Evaluation

```

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

proc shiftLeft(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ⊑ toGeneralNumber(a, phase);
  count: INTEGER ⊑ truncateToInteger(toGeneralNumber(b, phase));
  case x of
    FLOAT32 ⊑ FLOAT64 do
      i: {-231 ... 231 - 1} ⊑ signedWrap32(truncateToInteger(x));
      count ⊑ bitwiseAnd(count, 0x1F);
      i ⊑ signedWrap32(bitwiseShift(i, count));
      return realToFloat64(i);
    LONG do
      count ⊑ bitwiseAnd(count, 0x3F);
      i: {-263 ... 263 - 1} ⊑ signedWrap64(bitwiseShift(x.value, count));
      return LONG[value: i];
    ULONG do
      count ⊑ bitwiseAnd(count, 0x3F);
      i: {0 ... 264 - 1} ⊑ unsignedWrap64(bitwiseShift(x.value, count));
      return ULONG[value: i];
    end case
  end proc;

```

```

proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER  $\sqsubseteq$  toGeneralNumber(a, phase);
  count: INTEGER  $\sqsubseteq$  truncateToInteger(toGeneralNumber(b, phase));
  case x of
    FLOAT32  $\sqsubseteq$  FLOAT64 do
      i:  $\{-2^{31} \dots 2^{31} - 1\}$   $\sqsubseteq$  signedWrap32(truncateToInteger(x));
      count  $\sqsubseteq$  bitwiseAnd(count, 0x1F);
      i  $\sqsubseteq$  bitwiseShift(i, -count);
      return realToFloat64(i);
    LONG do
      count  $\sqsubseteq$  bitwiseAnd(count, 0x3F);
      i:  $\{-2^{63} \dots 2^{63} - 1\}$   $\sqsubseteq$  bitwiseShift(x.value, -count);
      return LONG[value: i]
    ULONG do
      count  $\sqsubseteq$  bitwiseAnd(count, 0x3F);
      i:  $\{-2^{63} \dots 2^{63} - 1\}$   $\sqsubseteq$  bitwiseShift(signedWrap64(x.value), -count);
      return ULONG[value: unsignedWrap64(i)]
  end case
end proc;

proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER  $\sqsubseteq$  toGeneralNumber(a, phase);
  count: INTEGER  $\sqsubseteq$  truncateToInteger(toGeneralNumber(b, phase));
  case x of
    FLOAT32  $\sqsubseteq$  FLOAT64 do
      i:  $\{0 \dots 2^{32} - 1\}$   $\sqsubseteq$  unsignedWrap32(truncateToInteger(x));
      count  $\sqsubseteq$  bitwiseAnd(count, 0x1F);
      i  $\sqsubseteq$  bitwiseShift(i, -count);
      return realToFloat64(i);
    LONG do
      count  $\sqsubseteq$  bitwiseAnd(count, 0x3F);
      i:  $\{0 \dots 2^{64} - 1\}$   $\sqsubseteq$  bitwiseShift(unsignedWrap64(x.value), -count);
      return LONG[value: signedWrap64(i)]
    ULONG do
      count  $\sqsubseteq$  bitwiseAnd(count, 0x3F);
      i:  $\{0 \dots 2^{64} - 1\}$   $\sqsubseteq$  bitwiseShift(x.value, -count);
      return ULONG[value: i]
  end case
end proc;

```

12.14 Relational Operators

Syntax

$\text{RelationalExpression}^{\text{allowIn}} \sqsubseteq$
 ShiftExpression
| $\text{RelationalExpression}^{\text{allowIn}} < \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} > \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \leq \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \geq \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \text{is} \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \text{as} \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \text{in} \text{ShiftExpression}$
| $\text{RelationalExpression}^{\text{allowIn}} \text{instanceof} \text{ShiftExpression}$

```

RelationalExpressionnoln □
| ShiftExpression
| RelationalExpressionnoln < ShiftExpression
| RelationalExpressionnoln > ShiftExpression
| RelationalExpressionnoln <= ShiftExpression
| RelationalExpressionnoln >= ShiftExpression
| RelationalExpressionnoln is ShiftExpression
| RelationalExpressionnoln as ShiftExpression
| RelationalExpressionnoln instanceof ShiftExpression

```

Validation

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

Setup

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

Evaluation

```

proc Eval[RelationalExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [RelationalExpression□ □ ShiftExpression] do
    return Eval[ShiftExpression](env, phase);
  [RelationalExpression0 □ RelationalExpression1 < ShiftExpression] do
    ra: OBJORREF □ Eval[RelationalExpression1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[ShiftExpression](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return isLess(a, b, phase);
  [RelationalExpression0 □ RelationalExpression1 > ShiftExpression] do
    ra: OBJORREF □ Eval[RelationalExpression1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[ShiftExpression](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return isLess(b, a, phase);
  [RelationalExpression0 □ RelationalExpression1 <= ShiftExpression] do
    ra: OBJORREF □ Eval[RelationalExpression1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[ShiftExpression](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return isLessOrEqual(a, b, phase);
  [RelationalExpression0 □ RelationalExpression1 >= ShiftExpression] do
    ra: OBJORREF □ Eval[RelationalExpression1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[ShiftExpression](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return isLessOrEqual(b, a, phase);
  [RelationalExpression□ □ RelationalExpression□ is ShiftExpression] do ???;
  [RelationalExpression□ □ RelationalExpression□ as ShiftExpression] do ???;
  [RelationalExpressionallowIn □ RelationalExpressionallowIn in ShiftExpression] do
    ???;

```

```

[RelationalExpression0 □ RelationalExpression1 instanceof ShiftExpression] do ???
end proc;

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

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

```

12.15 Equality Operators

Syntax

```

EqualityExpression0 □
  RelationalExpression0
  | EqualityExpression0 == RelationalExpression0
  | EqualityExpression0 != RelationalExpression0
  | EqualityExpression0 === RelationalExpression0
  | EqualityExpression0 !== RelationalExpression0

```

Validation

Validate[EqualityExpression⁰] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of EqualityExpression⁰.

Setup

Setup[EqualityExpression⁰] () propagates the call to Setup to every nonterminal in the expansion of EqualityExpression⁰.

Evaluation

```

proc Eval[EqualityExpression0] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [EqualityExpression0 □ RelationalExpression0] do
    return Eval[RelationalExpression0](env, phase);
  [EqualityExpression0_0 □ EqualityExpression0_1 == RelationalExpression0] do
    ra: OBJORREF □ Eval[EqualityExpression0_1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[RelationalExpression0](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return isEqual(a, b, phase);

```

```
[EqualityExpression0 □ EqualityExpression1 != RelationalExpression0] do
  ra: OBJORREF □ Eval[EqualityExpression1](env, phase);
  a: OBJECT □ readReference(ra, phase);
  rb: OBJORREF □ Eval[RelationalExpression0](env, phase);
  b: OBJECT □ readReference(rb, phase);
  c: BOOLEAN □ isEqual(a, b, phase);
  return not c;
[EqualityExpression0 □ EqualityExpression1 === RelationalExpression0] do
  ra: OBJORREF □ Eval[EqualityExpression1](env, phase);
  a: OBJECT □ readReference(ra, phase);
  rb: OBJORREF □ Eval[RelationalExpression0](env, phase);
  b: OBJECT □ readReference(rb, phase);
  return isStrictEqual(a, b, phase);
[EqualityExpression0 □ EqualityExpression1 != RelationalExpression0] do
  ra: OBJORREF □ Eval[EqualityExpression1](env, phase);
  a: OBJECT □ readReference(ra, phase);
  rb: OBJORREF □ Eval[RelationalExpression0](env, phase);
  b: OBJECT □ readReference(rb, phase);
  c: BOOLEAN □ isStrictEqual(a, b, phase);
  return not c
end proc;
```

```

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

proc isStrictlyEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  if a ∣ ALIASINSTANCE then return isStrictlyEqual(a.original, b, phase)
  elseif b ∣ ALIASINSTANCE then return isStrictlyEqual(a, b.original, phase)
  elseif a ∣ GENERALNUMBER and b ∣ GENERALNUMBER then
    return generalNumberCompare(a, b) = equal
  else return a = b
  end if
end proc;

```

12.16 Binary Bitwise Operators

Syntax

BitwiseAndExpression[□]
EqualityExpression[□]
 | *BitwiseAndExpression*[□] & *EqualityExpression*[□]

BitwiseXorExpression[□]
BitwiseAndExpression[□]
 | *BitwiseXorExpression*[□] ^ *BitwiseAndExpression*[□]

$$\begin{aligned} \text{BitwiseOrExpression}^{\square} &= \\ &\quad \text{BitwiseXorExpression}^{\square} \\ &\mid \text{BitwiseOrExpression}^{\square} \mid \text{BitwiseXorExpression}^{\square} \end{aligned}$$

Validation

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

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

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

Setup

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

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

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

Evaluation

```

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

proc Eval[BitwiseXorExpression□] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [BitwiseXorExpression□ □ BitwiseAndExpression□] do
    return Eval[BitwiseAndExpression□](env, phase);
  [BitwiseXorExpression□_0 □ BitwiseXorExpression□_1 ^ BitwiseAndExpression□] do
    ra: OBJORREF □ Eval[BitwiseXorExpression□_1](env, phase);
    a: OBJECT □ readReference(ra, phase);
    rb: OBJORREF □ Eval[BitwiseAndExpression□](env, phase);
    b: OBJECT □ readReference(rb, phase);
    return bitXor(a, b, phase)
  end proc;

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

```

```

[BitwiseOrExpression0 ⊕ BitwiseOrExpression1 | BitwiseXorExpression0] do
  ra: OBJORREF ⊕ Eval[BitwiseOrExpression1](env, phase);
  a: OBJECT ⊕ readReference(ra, phase);
  rb: OBJORREF ⊕ Eval[BitwiseXorExpression0](env, phase);
  b: OBJECT ⊕ readReference(rb, phase);
  return bitOr(a, b, phase)
end proc;

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
  x: GENERALNUMBER ⊕ toGeneralNumber(a, phase);
  y: GENERALNUMBER ⊕ toGeneralNumber(b, phase);
  if x ⊕ LONG ⊕ ULONG or y ⊕ LONG ⊕ ULONG then
    i: {-263 ... 263 - 1} ⊕ signedWrap64(truncateToInteger(x));
    j: {-263 ... 263 - 1} ⊕ signedWrap64(truncateToInteger(y));
    k: {-263 ... 263 - 1} ⊕ bitwiseAnd(i, j);
    if x ⊕ ULONG or y ⊕ ULONG then return ULONG[value: unsignedWrap64(k)]
    else return LONG[value: k]
    end if
  else
    i: {-231 ... 231 - 1} ⊕ signedWrap32(truncateToInteger(x));
    j: {-231 ... 231 - 1} ⊕ signedWrap32(truncateToInteger(y));
    return realToFloat64(bitwiseAnd(i, j))
  end if
end proc;

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
  x: GENERALNUMBER ⊕ toGeneralNumber(a, phase);
  y: GENERALNUMBER ⊕ toGeneralNumber(b, phase);
  if x ⊕ LONG ⊕ ULONG or y ⊕ LONG ⊕ ULONG then
    i: {-263 ... 263 - 1} ⊕ signedWrap64(truncateToInteger(x));
    j: {-263 ... 263 - 1} ⊕ signedWrap64(truncateToInteger(y));
    k: {-263 ... 263 - 1} ⊕ bitwiseXor(i, j);
    if x ⊕ ULONG or y ⊕ ULONG then return ULONG[value: unsignedWrap64(k)]
    else return LONG[value: k]
    end if
  else
    i: {-231 ... 231 - 1} ⊕ signedWrap32(truncateToInteger(x));
    j: {-231 ... 231 - 1} ⊕ signedWrap32(truncateToInteger(y));
    return realToFloat64(bitwiseXor(i, j))
  end if
end proc;

```

```

proc bitOr(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
  x: GENERALNUMBER  $\lceil$  toGeneralNumber(a, phase);
  y: GENERALNUMBER  $\lceil$  toGeneralNumber(b, phase);
  if x  $\lceil$  LONG  $\lceil$  ULONG or y  $\lceil$  LONG  $\lceil$  ULONG then
    i:  $\{-2^{63} \dots 2^{63} - 1\}$   $\lceil$  signedWrap64(truncateToInteger(x));
    j:  $\{-2^{63} \dots 2^{63} - 1\}$   $\lceil$  signedWrap64(truncateToInteger(y));
    k:  $\{-2^{63} \dots 2^{63} - 1\}$   $\lceil$  bitwiseOr(i, j);
    if x  $\lceil$  ULONG or y  $\lceil$  ULONG then return ULONG[value: unsignedWrap64(k)];
    else return LONG[value: k];
    end if
  else
    i:  $\{-2^{31} \dots 2^{31} - 1\}$   $\lceil$  signedWrap32(truncateToInteger(x));
    j:  $\{-2^{31} \dots 2^{31} - 1\}$   $\lceil$  signedWrap32(truncateToInteger(y));
    return realToFloat64(bitwiseOr(i, j));
  end if
end proc;

```

12.17 Binary Logical Operators

Syntax

```

LogicalAndExpression $\lceil$ 
  BitwiseOrExpression $\lceil$ 
  | LogicalAndExpression $\lceil$  && BitwiseOrExpression $\lceil$ 

LogicalXorExpression $\lceil$ 
  LogicalAndExpression $\lceil$ 
  | LogicalXorExpression $\lceil$  ^^ LogicalAndExpression $\lceil$ 

LogicalOrExpression $\lceil$ 
  LogicalXorExpression $\lceil$ 
  | LogicalOrExpression $\lceil$  || LogicalXorExpression $\lceil$ 

```

Validation

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

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

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

Setup

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

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

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

Evaluation

```

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

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

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

```

12.18 Conditional Operator

Syntax

```

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

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

```

Validation

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

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

Setup

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

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

Evaluation

```

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

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

```

12.19 Assignment Operators

Syntax

```

AssignmentExpression ◻
  ConditionalExpression ◻
  | PostfixExpression = AssignmentExpression ◻
  | PostfixExpression CompoundAssignment AssignmentExpression ◻
  | PostfixExpression LogicalAssignment AssignmentExpression ◻

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

LogicalAssignment ◻
  &&=
  ^^=
  ||=

```

Semantics

tag **andEq**;
tag **xorEq**;
tag **orEq**;

Validation

```

proc Validate[AssignmentExpression ◻] (ctx: CONTEXT, env: ENVIRONMENT)
  [AssignmentExpression ◻ ConditionalExpression ◻] do
    Validate[ConditionalExpression ◻](ctx, env);
  [AssignmentExpression ◻0 PostfixExpression = AssignmentExpression ◻1] do
    Validate[PostfixExpression](ctx, env);
    Validate[AssignmentExpression ◻](ctx, env);
  [AssignmentExpression ◻0 PostfixExpression CompoundAssignment AssignmentExpression ◻1] do
    Validate[PostfixExpression](ctx, env);
    Validate[AssignmentExpression ◻](ctx, env);
  [AssignmentExpression ◻0 PostfixExpression LogicalAssignment AssignmentExpression ◻1] do
    Validate[PostfixExpression](ctx, env);
    Validate[AssignmentExpression ◻](ctx, env)
end proc;

```

Setup

```
proc Setup[AssignmentExpression0]()
  [AssignmentExpression0 □ ConditionalExpression0] do Setup[ConditionalExpression0]()
  [AssignmentExpression0 □ PostfixExpression = AssignmentExpression1] do
    Setup[PostfixExpression]()
    Setup[AssignmentExpression1]()
  [AssignmentExpression0 □ PostfixExpression CompoundAssignment AssignmentExpression1] do
    Setup[PostfixExpression]()
    Setup[AssignmentExpression1]()
  [AssignmentExpression0 □ PostfixExpression LogicalAssignment AssignmentExpression1] do
    Setup[PostfixExpression]()
    Setup[AssignmentExpression1]()
end proc;
```

Evaluation

```
proc Eval[AssignmentExpression0] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [AssignmentExpression0 □ ConditionalExpression0] do
    return Eval[ConditionalExpression0](env, phase);
  [AssignmentExpression0 □ PostfixExpression = AssignmentExpression1] do
    if phase = compile then throw compileExpressionError end if;
    ra: OBJORREF □ Eval[PostfixExpression](env, phase);
    rb: OBJORREF □ Eval[AssignmentExpression1](env, phase);
    b: OBJECT □ readReference(rb, phase);
    writeReference(ra, b, phase);
    return b;
  [AssignmentExpression0 □ PostfixExpression CompoundAssignment AssignmentExpression1] do
    if phase = compile then throw compileExpressionError end if;
    rLeft: OBJORREF □ Eval[PostfixExpression](env, phase);
    oLeft: OBJECT □ readReference(rLeft, phase);
    rRight: OBJORREF □ Eval[AssignmentExpression1](env, phase);
    oRight: OBJECT □ readReference(rRight, phase);
    result: OBJECT □ Op[CompoundAssignment](oLeft, oRight, phase);
    writeReference(rLeft, result, phase);
    return result;
```

```

[AssignmentExpression0 □ PostfixExpression LogicalAssignment AssignmentExpression1] do
  if phase = compile then throw compileExpressionError end if;
  rLeft: OBJORREF □ Eval[PostfixExpression](env, phase);
  oLeft: OBJECT □ readReference(rLeft, phase);
  bLeft: BOOLEAN □ toBoolean(oLeft, phase);
  result: OBJECT □ oLeft;
  case Operator[LogicalAssignment] of
    {andEq} do
      if bLeft then
        result □ readReference(Eval[AssignmentExpression1](env, phase), phase)
      end if;
    {xorEq} do
      bRight: BOOLEAN □ toBoolean(readReference(Eval[AssignmentExpression1](env, phase), phase), phase);
      result □ bLeft xor bRight;
    {orEq} do
      if not bLeft then
        result □ readReference(Eval[AssignmentExpression1](env, phase), phase)
      end if;
    end case;
    writeReference(rLeft, result, phase);
    return result
  end proc;

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

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

```

12.20 Comma Expressions

Syntax

```

ListExpression0 □
  AssignmentExpression0
  | ListExpression0, AssignmentExpression0

```

```

OptionalExpression □
  ListExpressionallowIn
  | «empty»

```

Validation

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

Setup

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

Evaluation

```

proc Eval[ListExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ListExpression  $\sqcup$  AssignmentExpression] do
    return Eval[AssignmentExpression](env, phase);
  [ListExpression0  $\sqcup$  ListExpression1 , AssignmentExpression] do
    ra: OBJORREF  $\sqcup$  Eval[ListExpression1](env, phase);
    readReference(ra, phase);
    rb: OBJORREF  $\sqcup$  Eval[AssignmentExpression](env, phase);
    return readReference(rb, phase)
  end proc;

proc EvalAsList[ListExpression] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
  [ListExpression  $\sqcup$  AssignmentExpression] do
    r: OBJORREF  $\sqcup$  Eval[AssignmentExpression](env, phase);
    elt: OBJECT  $\sqcup$  readReference(r, phase);
    return [elt];
  [ListExpression0  $\sqcup$  ListExpression1 , AssignmentExpression] do
    elts: OBJECT[]  $\sqcup$  EvalAsList[ListExpression1](env, phase);
    r: OBJORREF  $\sqcup$  Eval[AssignmentExpression](env, phase);
    elt: OBJECT  $\sqcup$  readReference(r, phase);
    return elts  $\oplus$  [elt]
  end proc;

```

12.21 Type Expressions

Syntax

`TypeExpression` \sqcup `NonAssignmentExpression`

Validation

```

proc Validate[TypeExpression  $\sqcup$  NonAssignmentExpression] (ext: CONTEXT, env: ENVIRONMENT)
  Validate[NonAssignmentExpression](ext, env)
end proc;

```

Setup and Evaluation

```

proc SetupAndEval[TypeExpression  $\sqcup$  NonAssignmentExpression] (env: ENVIRONMENT): CLASS
  Setup[NonAssignmentExpression]();
  r: OBJORREF  $\sqcup$  Eval[NonAssignmentExpression](env, compile);
  o: OBJECT  $\sqcup$  readReference(r, compile);
  if o  $\sqcup$  CLASS then throw badValueError end if;
  return o
end proc;

```

13 Statements

Syntax

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

```

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

Enabled[Substatement□]: BOOLEAN;

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

proc Validate[Substatements] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
[Substatements □ «empty»] do nothing;
[Substatements □ SubstatementsPrefix Substatementabbrev] do
  Validate[SubstatementsPrefix](ctxt, env, jt);
  Validate[Substatementabbrev](ctxt, env, {}, jt)
end proc;

proc Validate[SubstatementsPrefix] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
[SubstatementsPrefix □ «empty»] do nothing;
[SubstatementsPrefix0 □ SubstatementsPrefix1 Substatementfull] do
  Validate[SubstatementsPrefix1](ctxt, env, jt);
  Validate[Substatementfull](ctxt, env, {}, jt)
end proc;

```

Setup

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

```

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

```

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

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

```

proc Setup[Semicolon□] ()
  [Semicolon□ □ ;] do nothing;
  [Semicolon□ □ VirtualSemicolon] do nothing;
  [Semicolonabbrev □ «empty»] do nothing;
  [SemicolonnoShortlf □ «empty»] do nothing
end proc;

```

Evaluation

```

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

proc Eval[Substatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Substatement□ □ EmptyStatement] do return d;
  [Substatement□ □ Statement□] do return Eval[Statement□](env, d);
  [Substatement□ □ SimpleVariableDefinition Semicolon□] do
    return Eval[SimpleVariableDefinition](env, d);

```

```

[Substatement□] Attributes [no line break] { Substatements } ] do
  if Enabled[Substatement□] then return Eval[Substatements](env, d)
  else return d
  end if
end proc;

proc Eval[Substatements] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Substatements] «empty»] do return d;
  [Substatements] SubstatementsPrefix Substatementabbrev] do
    o: OBJECT □ Eval[SubstatementsPrefix](env, d);
    return Eval[Substatementabbrev](env, o)
  end proc;

proc Eval[SubstatementsPrefix] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [SubstatementsPrefix] «empty»] do return d;
  [SubstatementsPrefix0] SubstatementsPrefix1 Substatementfull] do
    o: OBJECT □ Eval[SubstatementsPrefix1](env, d);
    return Eval[Substatementfull](env, o)
  end proc;

```

13.1 Empty Statement

Syntax

EmptyStatement □ ;

13.2 Expression Statement

Syntax

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

Validation

```

proc Validate[ExpressionStatement] □ [lookahead{function, {}}] ListExpressionallowIn
  (ctx: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpressionallowIn](ctx, env)
end proc;

```

Setup

```

proc Setup[ExpressionStatement] □ [lookahead{function, {}}] ListExpressionallowIn]()
  Setup[ListExpressionallowIn]()
end proc;

```

Evaluation

```

proc Eval[ExpressionStatement] □ [lookahead{function, {}}] ListExpressionallowIn] (env: ENVIRONMENT): OBJECT
  r: OBJORREF □ Eval[ListExpressionallowIn](env, run);
  return readReference(r, run)
end proc;

```

13.3 Super Statement

Syntax

SuperStatement □ **super** *Arguments*

Validation

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

Setup

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

Evaluation

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

13.4 Block Statement

Syntax

Block □ { *Directives* }

Validation

```
proc Validate[Block □ { Directives }] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
    compileFrame: BLOCKFRAME □
        new BLOCKFRAME[]$staticReadBindings: {}, staticWriteBindings: {}, plurality: pl[]|
            CompileFrame[Block] □ compileFrame;
            Validate[Directives](ext, [compileFrame] ⊕ env, jt, pl, none)
    end proc;

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

Setup

```
proc Setup[Block □ { Directives }] ()
    Setup[Directives]()
end proc;
```

Evaluation

```

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

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

CompileFrame[Block]: BLOCKFRAME;

```

13.5 Labeled Statements

Syntax

LabeledStatement □ *Identifier* : *Substatement*

Validation

```

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

```

Setup

```

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

```

Evaluation

```

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

```

13.6 If Statement

Syntax

```


$$\begin{aligned}
& \text{IfStatement}^{\text{abbrev}} \sqcup \\
& \quad \text{if ParenListExpression Substatement}^{\text{abbrev}} \\
| & \quad \text{if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{abbrev}} \\
\\
& \text{IfStatement}^{\text{full}} \sqcup \\
& \quad \text{if ParenListExpression Substatement}^{\text{full}} \\
| & \quad \text{if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{full}} \\
\\
& \text{IfStatement}^{\text{noShortIf}} \sqcup \text{ if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{noShortIf}}
\end{aligned}$$


```

Validation

```

proc Validate[IfStatement $\square$ ] (ctx: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
[IfStatement $\text{abbrev}$   $\sqcup$  if ParenListExpression Substatement $\text{abbrev}$ ] do
  Validate[ParenListExpression](ctx, env);
  Validate[Substatement $\text{abbrev}$ ](ctx, env, {}, jt);
[IfStatement $\text{full}$   $\sqcup$  if ParenListExpression Substatement $\text{full}$ ] do
  Validate[ParenListExpression](ctx, env);
  Validate[Substatement $\text{full}$ ](ctx, env, {}, jt);
[IfStatement $\square$   $\sqcup$  if ParenListExpression Substatement $\text{noShortIf}$ 1 else Substatement $\square$ 2] do
  Validate[ParenListExpression](ctx, env);
  Validate[Substatement $\text{noShortIf}$ 1](ctx, env, {}, jt);
  Validate[Substatement $\square$ 2](ctx, env, {}, jt)
end proc;

```

Setup

Setup[IfStatement \square] () propagates the call to Setup to every nonterminal in the expansion of IfStatement \square .

Evaluation

```

proc Eval[IfStatement $\square$ ] (env: ENVIRONMENT, d: OBJECT): OBJECT
[IfStatement $\text{abbrev}$   $\sqcup$  if ParenListExpression Substatement $\text{abbrev}$ ] do
  r: OBJORREF  $\sqcup$  Eval[ParenListExpression](env, run);
  o: OBJECT  $\sqcup$  readReference(r, run);
  if toBoolean(o, run) then return Eval[Substatement $\text{abbrev}$ ](env, d)
  else return d
  end if;
[IfStatement $\text{full}$   $\sqcup$  if ParenListExpression Substatement $\text{full}$ ] do
  r: OBJORREF  $\sqcup$  Eval[ParenListExpression](env, run);
  o: OBJECT  $\sqcup$  readReference(r, run);
  if toBoolean(o, run) then return Eval[Substatement $\text{full}$ ](env, d)
  else return d
  end if;
[IfStatement $\square$   $\sqcup$  if ParenListExpression Substatement $\text{noShortIf}$ 1 else Substatement $\square$ 2] do
  r: OBJORREF  $\sqcup$  Eval[ParenListExpression](env, run);
  o: OBJECT  $\sqcup$  readReference(r, run);
  if toBoolean(o, run) then return Eval[Substatement $\text{noShortIf}$ 1](env, d)
  else return Eval[Substatement $\square$ 2](env, d)
  end if
end proc;

```

13.7 Switch Statement

Syntax

SwitchStatement \sqsubseteq **switch** *ParenListExpression* { *CaseStatements* }

CaseStatements \sqsubseteq

 «empty»

 | *CaseLabel*

 | *CaseLabel CaseStatementsPrefix CaseStatement*^{abbrev}

CaseStatementsPrefix \sqsubseteq

 «empty»

 | *CaseStatementsPrefix CaseStatement*^{full}

CaseStatement^{full} \sqsubseteq

Substatement

 | *CaseLabel*

CaseLabel \sqsubseteq

case *ListExpression*^{allowIn} :

 | **default** :

Validation

```
proc Validate[SwitchStatement  $\sqsubseteq$  switch ParenListExpression { CaseStatements }]
  (ctxt: CONTEXT, env: ENVIRONMENT, jT: JUMPTARGETS)
  Validate[ParenListExpression](ctxt, env);
  *****
end proc;
```

Setup

```
proc Setup[SwitchStatement  $\sqsubseteq$  switch ParenListExpression { CaseStatements }]()
  *****
end proc;
```

Evaluation

```
proc Eval[SwitchStatement  $\sqsubseteq$  switch ParenListExpression { CaseStatements }]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
  *****
end proc;
```

13.8 Do-While Statement

Syntax

DoStatement \sqsubseteq **do** *Substatement*^{abbrev} **while** *ParenListExpression*

Validation

Labels[*DoStatement*]: LABEL {};

```

proc Validate[DoStatement □ do Substatementabbrev while ParenListExpression]
  (ctx: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
  continueLabels: LABEL{} □ sl □ {default};
  Labels[DoStatement] □ continueLabels;
  jt2: JUMPTARGETS □ JUMPTARGETS[breakTargets: jt.breakTargets □ {default},
    continueTargets: jt.continueTargets □ continueLabels];
  Validate[Substatementabbrev](ctx, env, {}, jt2);
  Validate[ParenListExpression](ctx, env)
end proc;

```

Setup

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

Evaluation

```

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

```

13.9 While Statement

Syntax

WhileStatement[□] □ **while** *ParenListExpression Substatement*[□]

Validation

```

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

```

Setup

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

Evaluation

```

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

13.10 For Statements

Syntax

```

ForStatement□  $\sqcap$ 
  for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□
  | for ( ForInBinding in ListExpressionallowIn ) Substatement□

ForInitialiser  $\sqcap$ 
  «empty»
  | ListExpressionnoln
  | VariableDefinitionKind VariableBindingListnoln
  | Attributes [no line break] VariableDefinitionKind VariableBindingListnoln

ForInBinding  $\sqcap$ 
  PostfixExpression
  | VariableDefinitionKind VariableBindingnoln
  | Attributes [no line break] VariableDefinitionKind VariableBindingnoln
```

Validation

```

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

Setup

```

proc Setup[ForStatement□] ()
  [ForStatement□  $\sqcap$  for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□]] do
    ???;
```

```
[ForStatement□] □ for ( ForInBinding in ListExpressionallowIn ) Substatement□] do  

    ????
end proc;
```

Evaluation

```
proc Eval[ForStatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT  

    [ForStatement□] □ for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement□] do  

        ????  

    [ForStatement□] □ for ( ForInBinding in ListExpressionallowIn ) Substatement□] do  

        ????  

end proc;
```

13.11 With Statement

Syntax

WithStatement[□] □ **with** ParenListExpression Substatement[□]

Validation

```
proc Validate[WithStatement□] □ with ParenListExpression Substatement□  

    (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)  

    Validate[ParenListExpression](ext, env);  

    Validate[Substatement□](ext, env, {}, jt)  

end proc;
```

Setup

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

Evaluation

```
proc Eval[WithStatement□] □ with ParenListExpression Substatement□] (env: ENVIRONMENT, d: OBJECT): OBJECT  

    ????  

end proc;
```

13.12 Continue and Break Statements

Syntax

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

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

Validation

```
proc Validate[ContinueStatement] (jt: JUMPTARGETS)  

    [ContinueStatement□] □ continue do  

        if default □ jt.continueTargets then throw syntaxError end if;
```

```

[ContinueStatement] [ continue [no line break] Identifier] do
  if Name[Identifier] [ jt.continueTargets then throw syntaxError end if
end proc;

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

```

Setup

```

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

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

```

Evaluation

```

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

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

```

13.13 Return Statement

Syntax

```

ReturnStatement [ return
| return [no line break] ListExpressionallowIn

```

Validation

```

proc Validate[ReturnStatement] (ctx: CONTEXT, env: ENVIRONMENT)
  [ReturnStatement] [ return] do
    if getRegionalFrame(env) [ PARAMETERFRAME then throw syntaxError end if;
  [ReturnStatement] [ return [no line break] ListExpressionallowIn] do
    if getRegionalFrame(env) [ PARAMETERFRAME then throw syntaxError end if;
    Validate[ListExpressionallowIn](ctx, env)
end proc;

```

Setup

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

Evaluation

```
proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
  [ReturnStatement [] return] do throw RETURNEDVALUE[value: undefined]
  [ReturnStatement [] return [no line break] ListExpressionallowIn] do
    r: OBJORREF [] Eval[ListExpressionallowIn](env, run);
    a: OBJECT [] readReference(r, run);
    throw RETURNEDVALUE[value: a]
  end proc;
```

13.14 Throw Statement

Syntax

`ThrowStatement [] throw [no line break] ListExpressionallowIn`

Validation

```
proc Validate[ThrowStatement [] throw [no line break] ListExpressionallowIn] (ctx: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpressionallowIn](ctx, env)
end proc;
```

Setup

```
proc Setup[ThrowStatement [] throw [no line break] ListExpressionallowIn] ()
  Setup[ListExpressionallowIn]()
end proc;
```

Evaluation

```
proc Eval[ThrowStatement [] throw [no line break] ListExpressionallowIn] (env: ENVIRONMENT): OBJECT
  r: OBJORREF [] Eval[ListExpressionallowIn](env, run);
  a: OBJECT [] readReference(r, run);
  throw THROWNVALUE[value: a]
end proc;
```

13.15 Try Statement

Syntax

`TryStatement []`
 `try Block CatchClauses`
 `| try Block CatchClausesOpt finally Block`

`CatchClausesOpt []`
 «empty»
 `| CatchClauses`

`CatchClauses []`
 `CatchClause`
 `| CatchClauses CatchClause`

`CatchClause [] catch (Parameter) Block`

Validation

```
proc Validate[TryStatement] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [TryStatement ┌ try Block CatchClauses] do
    Validate[Block](ctxt, env, jt, plural);
    Validate[CatchClauses](ctxt, env, jt);
  [TryStatement ┌ try Block1 CatchClausesOpt finally Block2] do
    Validate[Block1](ctxt, env, jt, plural);
    Validate[CatchClausesOpt](ctxt, env, jt);
    Validate[Block2](ctxt, env, jt, plural)
end proc;
```

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

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

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

Setup

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

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

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

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

Evaluation

```
proc Eval[TryStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [TryStatement ┌ try Block CatchClauses] do
    try return Eval[Block](env, d)
    catch x: SEMANTICEXCEPTION do
      if x ┌ THROWNVALUE then throw x end if;
      exception: OBJECT ┌ x.value;
      r: OBJECT ┌ {reject} ┌ Eval[CatchClauses](env, exception);
      if r ≠ reject then return r else throw x end if
    end try;
```

```
[TryStatement] try Block1 CatchClausesOpt finally Block2] do
  result: OBJECT | SEMANTICEXCEPTION;
  try result | Eval[Block1](env, d)
  catch x: SEMANTICEXCEPTION do result | x
  end try;
  if result | THROWNVALUE then
    exception: OBJECT | result.value;
    try
      r: OBJECT | {reject} | Eval[CatchClausesOpt](env, exception);
      if r ≠ reject then result | r end if
    catch y: SEMANTICEXCEPTION do result | y
    end try
  end if;
  Eval[Block2](env, undefined);
  case result of
    OBJECT do return result;
    SEMANTICEXCEPTION do throw result
  end case
end proc;

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

proc Eval[CatchClauses] (env: ENVIRONMENT, exception: OBJECT): OBJECT | {reject}
  [CatchClauses | CatchClause] do return Eval[CatchClause](env, exception);
  [CatchClauses0 | CatchClauses1 CatchClause] do
    r: OBJECT | {reject} | Eval[CatchClauses1](env, exception);
    if r ≠ reject then return r else return Eval[CatchClause](env, exception) end if
end proc;

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

14 Directives

Syntax

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

```

AnnotatableDirective□
| ExportDefinition Semicolon□
| VariableDefinition Semicolon□
| FunctionDefinition
| ClassDefinition
| NamespaceDefinition Semicolon□
| ImportDirective Semicolon□
| UseDirective Semicolon□

Directives□
  «empty»
  | DirectivesPrefix Directiveabbrev

DirectivesPrefix□
  «empty»
  | DirectivesPrefix Directivefull

```

Validation

```

proc Validate[Directive□] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
  attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
  [Directive□ EmptyStatement] do return ctxt;
  [Directive□ Statement□] do
    if attr □ {none, true} then throw syntaxError end if;
    Validate[Statement□](ctxt, env, {}, jt, pl);
    return ctxt;
  [Directive□ AnnotatableDirective□] do
    return Validate[AnnotatableDirective□](ctxt, env, pl, attr);
  [Directive□ Attributes [no line break] AnnotatableDirective□] do
    Validate[Attributes](ctxt, env);
    Setup[Attributes]();
    attr2: ATTRIBUTE □ Eval[Attributes](env, compile);
    attr3: ATTRIBUTE □ combineAttributes(attr, attr2);
    Enabled[Directive□] □ attr3 ≠ false;
    if attr3 ≠ false then return Validate[AnnotatableDirective□](ctxt, env, pl, attr3)
    else return ctxt
    end if;
  [Directive□ Attributes [no line break] { Directives } ] do
    Validate[Attributes](ctxt, env);
    Setup[Attributes]();
    attr2: ATTRIBUTE □ Eval[Attributes](env, compile);
    attr3: ATTRIBUTE □ combineAttributes(attr, attr2);
    Enabled[Directive□] □ attr3 ≠ false;
    if attr3 = false then return ctxt end if;
    return Validate[Directives](ctxt, env, jt, pl, attr3);
  [Directive□ PackageDefinition] do
    if attr □ {none, true} then ???? else throw syntaxError end if;
  [Directive□ Pragma Semicolon□] do
    if attr □ {none, true} then return Validate[Pragma](ctxt)
    else throw syntaxError
    end if
  end proc;

```

```

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

proc Validate[Directives] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
  attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
  [Directives □ «empty»] do return ctxt;
  [Directives □ DirectivesPrefix Directiveabbrev] do
    ctxt2: CONTEXT □ Validate[DirectivesPrefix](ctxt, env, jt, pl, attr);
    return Validate[Directiveabbrev](ctxt2, env, jt, pl, attr)
end proc;

proc Validate[DirectivesPrefix] (ctxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, pl: PLURALITY,
  attr: ATTRIBUTEOPTNOTFALSE): CONTEXT
  [DirectivesPrefix □ «empty»] do return ctxt;
  [DirectivesPrefix0 □ DirectivesPrefix1 Directivefull] do
    ctxt2: CONTEXT □ Validate[DirectivesPrefix1](ctxt, env, jt, pl, attr);
    return Validate[Directivefull](ctxt2, env, jt, pl, attr)
end proc;

```

Setup

```

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

```

```

proc Setup[AnnotatableDirective□] ()
  [AnnotatableDirective□ ExportDefinition Semicolon□] do ????;
  [AnnotatableDirective□ VariableDefinition Semicolon□] do
    Setup[VariableDefinition](0);
  [AnnotatableDirective□ FunctionDefinition] do Setup[FunctionDefinition](0);
  [AnnotatableDirective□ ClassDefinition] do Setup[ClassDefinition](0);
  [AnnotatableDirective□ NamespaceDefinition Semicolon□] do nothing;
  [AnnotatableDirective□ ImportDirective Semicolon□] do ????
  [AnnotatableDirective□ UseDirective Semicolon□] do nothing
end proc;

```

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

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

Evaluation

```

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

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

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

```

```

proc Eval[DirectivesPrefix] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [DirectivesPrefix □ «empty»] do return d;
  [DirectivesPrefix0 □ DirectivesPrefix1 Directivefull] do
    o: OBJECT □ Eval[DirectivesPrefix1](env, d);
    return Eval[Directivefull](env, o)
end proc;

Enabled[Directive□]: BOOLEAN;

```

14.1 Attributes

Syntax

Attributes □
 | *Attribute*
 | *AttributeCombination*

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

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

NonexpressionAttribute □
 | **final**
 | **private**
 | **static**

Validation

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

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

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

```

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

```

Setup

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

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

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

```
proc Setup[NonexpressionAttribute] ()  

  [NonexpressionAttribute  $\sqsubseteq$  final] do nothing;  

  [NonexpressionAttribute  $\sqsubseteq$  private] do nothing;  

  [NonexpressionAttribute  $\sqsubseteq$  static] do nothing  

end proc;
```

Evaluation

```
proc Eval[Attributes] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE  

  [Attributes  $\sqsubseteq$  Attribute] do return Eval[Attribute](env, phase);  

  [Attributes  $\sqsubseteq$  AttributeCombination] do return Eval[AttributeCombination](env, phase)  

end proc;  
  

proc Eval[AttributeCombination  $\sqsubseteq$  Attribute] [no line break] Attributes  

  (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE  

  a: ATTRIBUTE  $\sqsubseteq$  Eval[Attribute](env, phase);  

  if a = false then return false end if;  

  b: ATTRIBUTE  $\sqsubseteq$  Eval[Attributes](env, phase);  

  return combineAttributes(a, b)  

end proc;  
  

proc Eval[Attribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE  

  [Attribute  $\sqsubseteq$  AttributeExpression] do  

    r: OBJORREF  $\sqsubseteq$  Eval[AttributeExpression](env, phase);  

    a: OBJECT  $\sqsubseteq$  readReference(r, phase);  

    if a  $\sqsubseteq$  ATTRIBUTE then throw badValueError end if;  

    return a;  

  [Attribute  $\sqsubseteq$  true] do return true;  

  [Attribute  $\sqsubseteq$  false] do return false;  

  [Attribute  $\sqsubseteq$  public] do return publicNamespace;  

  [Attribute  $\sqsubseteq$  NonexpressionAttribute] do  

    return Eval[NonexpressionAttribute](env, phase)  

end proc;  
  

proc Eval[NonexpressionAttribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE  

  [NonexpressionAttribute  $\sqsubseteq$  final] do  

    return COMPOUNDATTRIBUTE[namespaces: {}, explicit: false, dynamic: false, memberMod: final,  

      overrideMod: none, prototype: false, unused: false]  

  [NonexpressionAttribute  $\sqsubseteq$  private] do  

    c: CLASSOPT  $\sqsubseteq$  getEnclosingClass(env);  

    Note that Validate ensured that c cannot be none at this point.  

    return c.privateNamespace;  

  [NonexpressionAttribute  $\sqsubseteq$  static] do  

    return COMPOUNDATTRIBUTE[namespaces: {}, explicit: false, dynamic: false, memberMod: static,  

      overrideMod: none, prototype: false, unused: false]  

end proc;
```

14.2 Use Directive

Syntax

UseDirective \sqsubseteq **use namespace** ParenListExpression

Validation

```

proc Validate[UseDirective  $\sqcap$  use namespace ParenListExpression] (ext: CONTEXT, env: ENVIRONMENT): CONTEXT
  Validate[ParenListExpression](ext, env);
  Setup[ParenListExpression] $\sqcap$ 0;
  values: OBJECT[]  $\sqcap$  EvalAsList[ParenListExpression] (env, compile);
  namespaces: NAMESPACE{} $\sqcap$  {};
  for each v  $\sqcap$  values do
    if v  $\sqcap$  NAMESPACE or v  $\sqcap$  namespaces then throw badValueError end if;
    namespaces  $\sqcap$  namespaces  $\sqcap$  {v}
  end for each;
  return CONTEXT[openNamespaces: ext.openNamespaces  $\sqcap$  namespaces, other fields from ext]
end proc;

```

14.3 Import Directive

Syntax

```

ImportDirective  $\sqcap$ 
  import ImportBinding IncludesExcludes
   $\sqcup$  import ImportBinding , namespace ParenListExpression IncludesExcludes

ImportBinding  $\sqcap$ 
  ImportSource
   $\sqcup$  Identifier = ImportSource

ImportSource  $\sqcap$ 
  String
   $\sqcup$  PackageName

IncludesExcludes  $\sqcap$ 
  «empty»
   $\sqcup$  , exclude ( NamePatterns )
   $\sqcup$  , include ( NamePatterns )

NamePatterns  $\sqcap$ 
  «empty»
   $\sqcup$  NamePatternList

NamePatternList  $\sqcap$ 
  QualifiedIdentifier
   $\sqcup$  NamePatternList , QualifiedIdentifier

```

14.4 Pragma

Syntax

```

Pragma  $\sqcap$  use PragmaItems

PragmaItems  $\sqcap$ 
  PragmaItem
   $\sqcup$  PragmaItems , PragmaItem

PragmaItem  $\sqcap$ 
  PragmaExpr
   $\sqcup$  PragmaExpr ?

```

```
PragmaExpr □
  Identifier
  | Identifier ( PragmaArgument )
```

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

Validation

```
proc Validate[Pragma □ use PragmaItems] (ctxt: CONTEXT): CONTEXT
  return Validate[PragmaItems](ctxt)
end proc;

proc Validate[PragmaItems] (ctxt: CONTEXT): CONTEXT
  [PragmaItems □ PragmaItem] do return Validate[PragmaItem](ctxt);
  [PragmaItems0 □ PragmaItems1, PragmaItem] do
    ctxt2: CONTEXT □ Validate[PragmaItems1](ctxt2);
    return Validate[PragmaItem](ctxt2)
  end proc;

  proc Validate[PragmaItem] (ctxt: CONTEXT): CONTEXT
    [PragmaItem □ PragmaExpr] do return Validate[PragmaExpr](ctxt, false);
    [PragmaItem □ PragmaExpr ?] do return Validate[PragmaExpr](ctxt, true)
  end proc;

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

    Value[PragmaArgument]: OBJECT;
    Value[PragmaArgument □ true] = true;
    Value[PragmaArgument □ false] = false;
    Value[PragmaArgument □ Number] = Value[Number];
    Value[PragmaArgument □ - Number] = generalNumberNegate(Value[Number]);
    Value[PragmaArgument □ - NegatedMinLong] = LONG[value: -263];
    Value[PragmaArgument □ String] = Value[String];
```

```

proc processPragma(ctxt: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN): CONTEXT
  if name = "strict" then
    if value ⊑ {true, undefined} then
      return CONTEXT[strict: true, other fields from ctxt]
    end if,
    if value = false then return CONTEXT[strict: false, other fields from ctxt] end if
  end if;
  if name = "ecmascript" then
    if value ⊑ {undefined, 4.0f64} then return ctxt end if;
    if value ⊑ {1.0f64, 2.0f64, 3.0f64} then
      An implementation may optionally modify ctxt to disable features not available in ECMAScript Edition value
      other than subsequent pragmas.
      return ctxt
    end if
  end if;
  if optional then return ctxt else throw badValueError end if
end proc;

```

15 Definitions

15.1 Export Definition

Syntax

ExportDefinition ⊑ **export** *ExportBindingList*

ExportBindingList ⊑
ExportBinding
| *ExportBindingList* , *ExportBinding*

ExportBinding ⊑
FunctionName
| *FunctionName* = *FunctionName*

15.2 Variable Definition

Syntax

VariableDefinition ⊑ *VariableDefinitionKind* *VariableBindingList*^{allowIn}

VariableDefinitionKind ⊑
var
| **const**

*VariableBindingList*⁰ ⊑
*VariableBinding*⁰
| *VariableBindingList*⁰ , *VariableBinding*⁰

*VariableBinding*⁰ ⊑ *TypedIdentifier*⁰ *VariableInitialisation*⁰

*VariableInitialisation*⁰ ⊑
«empty»
| = *VariableInitialiser*⁰

```

VariableInitialiser□
| AssignmentExpression□
| NonexpressionAttribute
| AttributeCombination

TypedIdentifier□
| Identifier
| Identifier : TypeExpression□

```

Validation

```

proc Validate[VariableDefinition □ VariableDefinitionKind VariableBindingListallowIn]
  (ctx: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE)
  immutable: BOOLEAN □ Immutable[VariableDefinitionKind];
  Validate[VariableBindingListallowIn](ctx, env, attr, immutable)
end proc;  

  

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

  

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

```

```

CompileEnv[VariableBinding□]: ENVIRONMENT;
CompileVar[VariableBinding□]: HOISTEDVAR □ VARIABLE □ INSTANCEVARIABLE;
OverriddenRead[VariableBinding□]: OVERRIDDENMEMBER;
OverriddenWrite[VariableBinding□]: OVERRIDDENMEMBER;
Multiname[VariableBinding□]: MULTINAME;

```

```

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

```

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

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

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

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

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

Setup

```
proc Setup[VariableDefinition □ VariableDefinitionKind VariableBindingListallowIn] ()
  Setup[VariableBindingListallowIn]()
```

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

```

proc Setup[VariableBinding□ TypedIdentifier□ VariableInitialisation□]()
  env: ENVIRONMENT □ CompileEnv[VariableBinding□];
  v: HOISTEDVAR □ VARIABLE □ INSTANCEVARIABLE □ CompileVar[VariableBinding□];
  case v of
    HOISTEDVAR do Setup[VariableInitialisation□]( );
    VARIABLE do
      type: CLASS □ getVariableType(v, compile);
      case v.value of
        OBJECT do nothing;
        {inaccessible} do Setup[VariableInitialisation□]( );
        () □ OBJECT do
          v.value □ inaccessible;
          Setup[VariableInitialisation□]( );
        try
          value: OBJECTOPT □ Eval[VariableInitialisation□](env, compile);
          if value  $\neq$  none then
            coercedValue: OBJECT □ type.implicitCoerce(value);
            v.value □ coercedValue
          end if
        catch x: SEMANTICEXCEPTION do
          if x  $\neq$  compileExpressionError then throw x end if;
          If a compileExpressionError occurred, then the initialiser is not a compile-time constant expression.
          In this case, ignore the error and leave the value of the variable inaccessible until it is defined at run time.
        end try
      end case;
    INSTANCEVARIABLE do
      t: CLASSOPT □ SetupAndEval[TypedIdentifier□](env);
      if t = none then
        overriddenRead: OVERRIDDENMEMBER □ OverriddenRead[VariableBinding□];
        overriddenWrite: OVERRIDDENMEMBER □ OverriddenWrite[VariableBinding□];
        if overriddenRead □ {none, potentialConflict} then
          Note that defineInstanceMember already ensured that overriddenRead □ INSTANCEMETHOD.
          t □ overriddenRead.type
        elsif overriddenWrite □ {none, potentialConflict} then
          Note that defineInstanceMember already ensured that overriddenWrite □ INSTANCEMETHOD.
          t □ overriddenWrite.type
        else t □ objectClass
        end if
      end if;
      v.type □ t;
      Setup[VariableInitialisation□]( )
    end case
  end proc;

```

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

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

Evaluation

```

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

proc Eval[VariableBindingList□] (env: ENVIRONMENT, immutable: BOOLEAN)
  [VariableBindingList□ □ VariableBinding□] do Eval[VariableBinding□](env, immutable);
  [VariableBindingList□ □ VariableBindingList□1 , VariableBinding□] do
    Eval[VariableBindingList□1](env, immutable);
    Eval[VariableBinding□](env, immutable)
end proc;

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

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

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

```

```

[VariableInitialiser□ □ AttributeCombination] do
    return Eval[AttributeCombination](env, phase)
end proc;

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

```

15.3 Simple Variable Definition

Syntax

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

SimpleVariableDefinition □ **var** *UntypedVariableBindingList*

UntypedVariableBindingList □
UntypedVariableBinding
| *UntypedVariableBindingList* , *UntypedVariableBinding*

UntypedVariableBinding □ *Identifier* *VariableInitialisation*^{allowIn}

Validation

```

proc Validate[SimpleVariableDefinition □ var UntypedVariableBindingList] (ctx: CONTEXT, env: ENVIRONMENT)
    if ctx.strict or getRegionalFrame(env) □ GLOBAL □ PARAMETERFRAME then
        throw syntaxError
    end if;
    Validate[UntypedVariableBindingList](ctx, env)
end proc;

```

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

```

proc Validate[UntypedVariableBinding □ Identifier VariableInitialisationallowIn] (ctx: CONTEXT, env: ENVIRONMENT)
    Validate[VariableInitialisationallowIn](ctx, env);
    defineHoistedVar(env, Name[Identifier], undefined)
end proc;

```

Setup

```

proc Setup[SimpleVariableDefinition □ var UntypedVariableBindingList] ()
    Setup[UntypedVariableBindingList]()
end proc;

```

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

```

proc Setup[UntypedVariableBinding □ Identifier VariableInitialisationallowIn] ()
    Setup[VariableInitialisationallowIn]()
end proc;

```

Evaluation

```

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

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

proc Eval[UntypedVariableBinding ⊑ Identifier VariableInitialisationallowIn] (env: ENVIRONMENT)
  value: OBJECTOPT ⊑ Eval[VariableInitialisationallowIn](env, run);
  if value ≠ none then
    qname: QUALIFIEDNAME ⊑ QUALIFIEDNAME[Namespace: publicNamespace, id: Name[Identifier]] ⊑
    lexicalWrite(env, {qname}, value, false, run)
  end if
end proc;

```

15.4 Function Definition

Syntax

FunctionDefinition ⊑ **function** *FunctionName* *FunctionCommon*

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

FunctionCommon ⊑ (*Parameters*) *Result Block*

Validation

```

proc Validate[FunctionDefinition  $\sqsubseteq$  function FunctionName FunctionCommon]
  (ctxt: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOPTNOTFALSE)
  name: STRING  $\sqsubseteq$  Name[FunctionName];
  kind: FUNCTIONKIND  $\sqsubseteq$  Kind[FunctionName];
  a: COMPOUNDATTRIBUTE  $\sqsubseteq$  toCompoundAttribute(attr);
  if a.dynamic then throw definitionError end if;
  unchecked: BOOLEAN  $\sqsubseteq$  not ctxt.strict and env[0]  $\sqsubseteq$  CLASS and kind = normal and Untyped[FunctionCommon];
  Unchecked[FunctionCommon]  $\sqsubseteq$  unchecked;
  prototype: BOOLEAN  $\sqsubseteq$  unchecked or a.prototype;
  memberMod: MEMBERMODIFIER  $\sqsubseteq$  a.memberMod;
  if env[0]  $\sqsubseteq$  CLASS then if memberMod = none then memberMod  $\sqsubseteq$  virtual end if
  else if memberMod  $\neq$  none then throw definitionError end if
  end if;
  if prototype and (kind  $\neq$  normal or memberMod = constructor) then
    throw definitionError
  end if;
  this: {none, inaccessible}  $\sqsubseteq$  none;
  if prototype or memberMod  $\sqsubseteq$  {constructor, virtual, final} then this  $\sqsubseteq$  inaccessible
  end if;
  case memberMod of
    {none, static} do
      f: INSTANCE  $\sqsubseteq$  OPENINSTANCE;
      if kind  $\sqsubseteq$  {get, set} then ????
      else f  $\sqsubseteq$  ValidateStaticFunction[FunctionCommon](ctxt, env, this, prototype)
      end if;
      if pl = singular then f  $\sqsubseteq$  instantiateOpenInstance(f, env) end if;
      if unchecked and attr = none and
        (env[0]  $\sqsubseteq$  GLOBAL or (env[0]  $\sqsubseteq$  BLOCKFRAME and env[0]  $\sqsubseteq$  PARAMETERFRAME)) then
          defineHoistedVar(env, name, f)
        else
          v: VARIABLE  $\sqsubseteq$  new VARIABLE[Type: functionClass, value: f, immutable: true];
          defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
        end if;
    {virtual, final} do ????
    {constructor} do ???
  end case
end proc;

```

Kind[*FunctionName*]: FUNCTIONKIND;
 Kind[*FunctionName* \sqsubseteq Identifier] = normal;
 Kind[*FunctionName* \sqsubseteq get [no line break] Identifier] = get;
 Kind[*FunctionName* \sqsubseteq set [no line break] Identifier] = set;

Name[*FunctionName*]: STRING;
 Name[*FunctionName* \sqsubseteq Identifier] = Name[Identifier];
 Name[*FunctionName* \sqsubseteq get [no line break] Identifier] = Name[Identifier];
 Name[*FunctionName* \sqsubseteq set [no line break] Identifier] = Name[Identifier];

Untyped[*FunctionCommon* \sqsubseteq (Parameters) Result Block]: BOOLEAN = Untyped[Parameters] **and** Untyped[Result];

Unchecked[*FunctionCommon*]: BOOLEAN;

CompileEnv[*FunctionCommon*]: ENVIRONMENT;

```

CompileFrame[FunctionCommon]: PARAMETERFRAME;
Signature[FunctionCommon]: SIGNATURE;

proc Validate[FunctionCommon □ ( Parameters ) Result Block]
  (ctx: CONTEXT, env: ENVIRONMENT, this: {none, inaccessible}, prototype: BOOLEAN): INTEGER
  compileFrame: PARAMETERFRAME □ new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}],
    plurality: plural, this: this, prototype: prototype[][]
  compileEnv: ENVIRONMENT □ [compileFrame] ⊕ env;
  CompileFrame[FunctionCommon] □ compileFrame;
  CompileEnv[FunctionCommon] □ compileEnv;
  nFixedParameters: INTEGER □ Validate[Parameters](ctx, compileEnv);
  Validate[Result](ctx, compileEnv);
  Validate[Block](ctx, compileEnv, JUMPTARGETS[breakTargets: {}, continueTargets: {}] plural);
  return nFixedParameters
end proc;

proc ValidateStaticFunction[FunctionCommon □ ( Parameters ) Result Block]
  (ctx: CONTEXT, env: ENVIRONMENT, this: {none, inaccessible}, prototype: BOOLEAN): OPENINSTANCE
  nFixedParameters: INTEGER □ Validate[FunctionCommon](ctx, env, this, prototype);
  if prototype then ???
  else
    initialSlots: SLOT{} □ {new SLOT[] id: findInstanceMember(functionClass,
      QUALIFIEDNAME[namespace: publicNamespace, id: "length"] read),
      value: realToFloat64(nFixedParameters)}[];
    return new OPENINSTANCE[] type: functionClass, typeofString: "Function", defaultSlots: initialSlots,
      buildPrototype: false, call: EvalNormalCall[FunctionCommon], construct: badConstruct, cache: none[]
  end if
end proc;

```

Setup

```

proc Setup[FunctionDefinition □ function FunctionName FunctionCommon] ()
  Setup[FunctionCommon]()
end proc;

proc Setup[FunctionCommon □ ( Parameters ) Result Block] ()
  ???
end proc;

```

Evaluation

```

proc EvalNormalCall[FunctionCommon □ ( Parameters ) Result Block]
  (this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
  if phase = compile then throw compileExpressionError end if;
  runtimeFrame: PARAMETERFRAME □ new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}],
    plurality: singular, this: none, prototype: false[]
  instantiateFrame(CompileFrame[FunctionCommon], runtimeFrame, [runtimeFrame] ⊕ runtimeEnv);
  assignArguments(runtimeFrame, Signature[FunctionCommon], Unchecked[FunctionCommon], args);
  result: OBJECT;
  try Eval[Block](runtimeFrame] ⊕ runtimeEnv, undefined), result □ undefined
  catch x: SEMANTICEXCEPTION do
    if x □ RETURNEDVALUE then result □ x.value else throw x end if
  end try;
  return result
end proc;

```

```

proc EvalPrototypeCall[FunctionCommon □ ( Parameters ) Result Block]
  (this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
  if phase = compile then throw compileExpressionError end if;
  runtimeThis: OBJECT □ this;
  g: PACKAGE □ GLOBAL □ getPackageOrGlobalFrame(runtimeEnv);
  if runtimeThis □ {null, undefined} and g □ GLOBAL then runtimeThis □ g end if;
  runtimeFrame: PARAMETERFRAME □ new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}, plurality: singular, this: runtimeThis, prototype: true];
  instantiateFrame(CompileFrame[FunctionCommon], runtimeFrame, [runtimeFrame] ⊕ runtimeEnv);
  assignArguments(runtimeFrame, Signature[FunctionCommon], Unchecked[FunctionCommon], args);
  result: OBJECT;
  try Eval[Block]([runtimeFrame] ⊕ runtimeEnv, undefined); result □ undefined
  catch x: SEMANTICEXCEPTION do
    if x □ RETURNEDVALUE then result □ x.value else throw x end if
  end try;
  return result
end proc;

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

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

```

Syntax

```

Parameters □
  «empty»
  | AllParameters

AllParameters □
  Parameter
  | Parameter , AllParameters
  | OptionalParameters

OptionalParameters □
  OptionalParameter
  | OptionalParameter , OptionalParameters
  | RestAndNamedParameters

RestAndNamedParameters □
  NamedParameters
  | RestParameter
  | RestParameter , NamedParameters
  | NamedRestParameter

NamedParameters □
  NamedParameter
  | NamedParameter , NamedParameters

Parameter □
  TypedIdentifierallowIn
  | const TypedIdentifierallowIn

```

```

OptionalParameter □ Parameter = AssignmentExpressionallowIn
TypedInitialiser □ TypedIdentifierallowIn = AssignmentExpressionallowIn
NamedParameter □
  | named TypedInitialiser
  | const named TypedInitialiser
  | named const TypedInitialiser

RestParameter □
  ...
  | ... Parameter

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

Result □
  «empty»
  | : TypeExpressionallowIn

```

Validation

```

Untyped[Parameters]: BOOLEAN;
Untyped[Parameters □ «empty»] = true;
Untyped[Parameters □ AllParameters] = ????;

proc Validate[Parameters] (ctxt: CONTEXT, env: ENVIRONMENT): INTEGER
  [Parameters □ «empty»] do return 0;
  [Parameters □ AllParameters] do ??????
end proc;

Untyped[Result]: BOOLEAN;
Untyped[Result □ «empty»] = true;
Untyped[Result □ : TypeExpressionallowIn] = false;
```

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

15.5 Class Definition

Syntax

```

ClassDefinition □ class Identifier Inheritance Block
Inheritance □
  «empty»
  | extends TypeExpressionallowIn

```

Validation

```
Class[ClassDefinition]: CLASS;
```

```

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

proc Validate[Inheritance] (ext: CONTEXT, env: ENVIRONMENT): CLASS
  [Inheritance □ «empty»] do return objectClass;
  [Inheritance □ extends TypeExpressionallowIn] do
    Validate[TypeExpressionallowIn](ext, env);
    return SetupAndEval[TypeExpressionallowIn](env)
  end proc;

```

Setup

```

proc Setup[ClassDefinition □ class Identifier Inheritance Block]()
  Setup[Block]()
end proc;

```

Evaluation

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

15.6 Namespace Definition

Syntax

NamespaceDefinition □ **namespace** *Identifier*

Validation

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

15.7 Package Definition

Syntax

PackageDefinition □
 | **package** *Block*
 | **package** *PackageName* *Block*

PackageName □
 | *Identifier*
 | *PackageName* . *Identifier*

16 Programs

Syntax

Program □ *Directives*

Evaluation

```

EvalProgram[Program ⊕ Directives]: OBJECT
begin
  Validate[Directives](initialContext, initialEnvironment, JUMPTARGETS[], breakTargets: {}, continueTargets: {} ⊕
    singular, none);
  Setup[Directives]();
  return Eval[Directives](initialEnvironment, undefined)
end;

```

17 Predefined Identifiers

18 Built-in Classes

```

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

objectClass: CLASS = makeBuiltInClass(none, false, true, false, undefined);
undefinedClass: CLASS = makeBuiltInClass(objectClass, false, false, true, undefined);
nullClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
booleanClass: CLASS = makeBuiltInClass(objectClass, false, false, true, false);
generalNumberClass: CLASS = makeBuiltInClass(objectClass, false, false, false, NaN64f64);
longClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, LONG[value: 0]);
uLongClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, ULONG[value: 0]);
floatClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, NaN32f32);
numberClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, NaN64f64);
characterClass: CLASS = makeBuiltInClass(objectClass, false, false, true, «NUL»);
stringClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
namespaceClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);

```

```
attributeClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
classClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
functionClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
prototypeClass: CLASS = makeBuiltInClass(objectClass, true, true, true, null);
packageClass: CLASS = makeBuiltInClass(objectClass, true, true, true, null);
objectPrototype: PROTOTYPE = new PROTOTYPE[] parent: none, dynamicProperties: {}[]
```

18.1 Object

18.2 Never

18.3 Void

18.4 Null

18.5 Boolean

18.6 Integer

18.7 Number

18.7.1 ToNumber Grammar

18.8 Character

18.9 String

18.10 Function

18.11 Array

18.12 Type

18.13 Math

18.14 Date

18.15 RegExp

18.15.1 Regular Expression Grammar

18.16 Error

18.17 Attribute

19 Built-in Functions

20 Built-in Attributes

21 Built-in Namespaces

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

A Index

A.1 Nonterminals

<i>AdditiveExpression</i>	97	<i>DecimalIntegerLiteral</i>	32	<i>IdentifierOrKeyword</i>	29
<i>AllParameters</i>	145	<i>DecimalLiteral</i>	32	<i>IdentityEscape</i>	34
<i>AnnotatableDirective</i>	128	<i>Directive</i>	127	<i>IfStatement</i>	118
<i>Arguments</i>	90	<i>Directives</i>	128	<i>ImportBinding</i>	133
<i>ArrayLiteral</i>	83	<i>DirectivesPrefix</i>	128	<i>ImportDirective</i>	133
<i>ASCIIDigit</i>	32	<i>DivisionPunctuator</i>	31	<i>ImportSource</i>	133
<i>AssignmentExpression</i>	109	<i>DoStatement</i>	120	<i>IncludesExcludes</i>	133
<i>Attribute</i>	131	<i>ElementList</i>	83	<i>Inheritance</i>	146
<i>AttributeCombination</i>	131	<i>EmptyStatement</i>	116	<i>InitialIdentifierCharacter</i>	30
<i>AttributeExpression</i>	85	<i>EndOfInput</i>	27	<i>InitialIdentifierCharacterOrEscape</i>	30
<i>Attributes</i>	131	<i>EqualityExpression</i>	102	<i>InputElement</i>	27
<i>BitwiseAndExpression</i>	104	<i>ExportBinding</i>	135	<i>IntegerLiteral</i>	31
<i>BitwiseOrExpression</i>	104	<i>ExportBindingList</i>	135	<i>LabeledStatement</i>	118
<i>BitwiseXorExpression</i>	104	<i>ExportDefinition</i>	135	<i>LetterE</i>	32
<i>Block</i>	117	<i>ExpressionQualifiedIdentifier</i>	77	<i>LetterF</i>	31
<i>BlockCommentCharacters</i>	28	<i>ExpressionStatement</i>	116	<i>LetterL</i>	31
<i>Brackets</i>	90	<i>FieldList</i>	81	<i>LetterU</i>	31
<i>BreakStatement</i>	123	<i>FieldName</i>	81	<i>LetterX</i>	32
<i>CaseLabel</i>	120	<i>ForInBinding</i>	122	<i>LineBreak</i>	28
<i>CaseStatement</i>	120	<i>ForInitialiser</i>	122	<i>LineBreaks</i>	28
<i>CaseStatements</i>	119	<i>ForStatement</i>	122	<i>LineComment</i>	28
<i>CaseStatementsPrefix</i>	119	<i>Fraction</i>	32	<i>LineCommentCharacters</i>	28
<i>CatchClause</i>	125	<i>FullNewExpression</i>	85	<i>LineTerminator</i>	28
<i>CatchClauses</i>	125	<i>FullNewSubexpression</i>	85	<i>ListExpression</i>	111
<i>CatchClausesOpt</i>	125	<i>FullPostfixExpression</i>	85	<i>LiteralElement</i>	83
<i>ClassDefinition</i>	146	<i>FunctionCommon</i>	142	<i>LiteralField</i>	81
<i>CompoundAssignment</i>	109	<i>FunctionDefinition</i>	142	<i>LiteralStringChar</i>	34
<i>ConditionalExpression</i>	108	<i>FunctionExpression</i>	81	<i>LogicalAndExpression</i>	106
<i>ContinueStatement</i>	123	<i>FunctionName</i>	142	<i>LogicalAssignment</i>	109
<i>ContinuingIdentifierCharacter</i>	30	<i>HexDigit</i>	32	<i>LogicalOrExpression</i>	106
<i>ContinuingIdentifierCharacterOrEsca</i>	pe 30	<i>HexEscape</i>	34	<i>LogicalXorExpression</i>	106
<i>ControlEscape</i>	34	<i>HexIntegerLiteral</i>	32	<i>Mantissa</i>	32
<i>DecimalDigits</i>	32	<i>Identifier</i>	77	<i>MemberOperator</i>	90
		<i>IdentifierName</i>	29	<i>MultiLineBlockComment</i>	29

<i>MultiLineBlockCommentCharacters</i>	85	<i>StringEscape</i>	34
29		<i>StringLiteral</i>	34
<i>MultiplicativeExpression</i>	95	<i>Substatement</i>	113
<i>NamedArgumentList</i>	90	<i>Substatements</i>	113
<i>NamedParameter</i>	146	<i>SubstatementsPrefix</i>	113
<i>NamedParameters</i>	145	<i>SuperExpression</i>	84
<i>NamedRestParameter</i>	146	<i>SuperStatement</i>	116
<i>NamePatternList</i>	133	<i>SwitchStatement</i>	119
<i>NamePatterns</i>	133	<i>ThrowStatement</i>	125
<i>NamespaceDefinition</i>	148	<i>TryStatement</i>	125
<i>NextInputElement</i>	27	<i>TypedIdentifier</i>	136
<i>NonAssignmentExpression</i>	108	<i>TypedInitialiser</i>	146
<i>NonexpressionAttribute</i>	131	<i>TypeExpression</i>	112
<i>NonTerminator</i>	29	<i>UnaryExpression</i>	92
<i>NonTerminatorOrAsteriskOrSlash</i>	29	<i>UnicodeAlphanumeric</i>	30
<i>NonTerminatorOrSlash</i>	29	<i>UnicodeCharacter</i>	29
<i>NonZeroDecimalDigits</i>	32	<i>UntypedVariableBinding</i>	141
<i>NonZeroDigit</i>	32	<i>UntypedVariableBindingList</i>	141
<i>NullEscape</i>	30	<i>UseDirective</i>	132
<i>NullEscapes</i>	30	<i>VariableBinding</i>	135
<i>NumericLiteral</i>	31	<i>VariableBindingList</i>	135
<i>ObjectLiteral</i>	81	<i>VariableDefinition</i>	135
<i>OptionalExpression</i>	111	<i>VariableDefinitionKind</i>	135
<i>OptionalParameter</i>	146	<i>VariableInitialisation</i>	135
<i>OptionalParameters</i>	145	<i>VariableInitialiser</i>	136
<i>OrdinaryRegExpChar</i>	36	<i>WhileStatement</i>	121
<i>PackageDefinition</i>	148	<i>WhiteSpace</i>	28
<i>PackageName</i>	148	<i>WhiteSpaceCharacter</i>	28
<i>Parameter</i>	145	<i>WithStatement</i>	123
<i>Parameters</i>	145	<i>ZeroEscape</i>	34
<i>ParenExpression</i>	79		
<i>ParenExpressions</i>	90		
<i>ParenListExpression</i>	79		

A.2 Tags

-• 10, 12
+• 10, 12
+zero 10, 12
abstract 38, 48
andEq 109
compile 45
constructor 38
default 45
equal 51
false 4, 37
final 38
fixed 41
forbidden 47
generic 68

get 44
greater 51
inaccessible 37, 45
less 51
NaN 10, 12
none 37, 38, 39, 40
normal 44
null 37
orEq 109
plural 45
potentialConflict 61
propertyLookup 65
read 58
readWrite 58

run 45
set 44
singular 45
static 38
true 4, 37
undefined 37
uninitialised 37
unordered 51
virtual 38
write 58
xorEq 109
-zero 10, 12

A.3 Semantic Domains

ACCESS 58
ALIASINSTANCE 41
ARGUMENTLIST 44
ATTRIBUTE 39

ATTRIBUTEOPTNOTFALSE 39
BLOCKFRAME 46
BOOLEAN 4, 37
BOOLEANOPT 37

BRACKETREFERENCE 43
CALLABLEINSTANCE 41
CHARACTER 7
CLASS 39

COMPOUNDATTRIBUTE 38
CONSTRUCTORMETHOD 47
CONTEXT 45
DENORMALISEDFLOAT32VALUES 11
DENORMALISEDFLOAT64VALUES 12
DOTREFERENCE 43
 41
DYNAMICOBJECT 37
DYNAMICPROPERTY 40
ENVIRONMENT 45
ENVIRONMENTI 45
FINITEFLOAT32 11
FINITEFLOAT64 12
FINITEGENERALNUMBER 10
 40
FLOAT32 10
FLOAT64 12
FRAME 45
FUNCTIONKIND 44
GENERALNUMBER 10
GLOBAL 42
HOISTEDVAR 47
INPUTELEMENT 26
INSTANCE 40
INSTANCEBINDING 48
INSTANCEGETTER 48
INSTANCEMEMBER 48
INSTANCEMEMBEROPT 48
INSTANCEMETHOD 48
INSTANCESETTER 48
INSTANCEVARIABLE 48
INTEGER 6
INTEGEROPT 37
JUMPTARGETS 45
LABEL 45
LEXICALLOOKUP 66
LEXICALREFERENCE 43
LIMITEDINSTANCE 42
LONG 10
LOOKUPKIND 66
MEMBERINSTANTIATION 64
MEMBERMODIFIER 38
METHODCLOSURE 40
MULTINAME 38
NAMEDARGUMENT 44
NAMEDPARAMETER 44
NAMESPACE 38
 40
NONZEROFINITEFLOAT32 11
NONZEROFINITEFLOAT64 12
NORMALISEDFLOAT32VALUES 11
NORMALISEDFLOAT64VALUES 12
NULL 37
OBJECT 37
OBJECTI 37
OBJECTIOPT 37
OBJECTOPT 37
OBJECTU 37
OBJOPTIONALLIMIT 43
OBJORREF 43
OPENINSTANCE 41
ORDER 51
OVERRIDDENMEMBER 61
OVERRIDE MODIFIER 38
OVERRIDESTATUS 61
OVERRIDESTATUSPAIR 61
PACKAGE 42
PARAMETER 44
PARAMETERFRAME 46
PHASE 45
PLURALITY 45
PRIMITIVEOBJECT 37
PROTOTYPE 40
PROTOTYPEOPT 40
QUALIFIEDNAME 38
QUALIFIEDNAMEOPT 38
RATIONAL 6
REAL 6
REFERENCE 43
SETTER 47
SIGNATURE 44
SIMPLEINSTANCE 40
SLOT 42
STATICBINDING 46
STATICMEMBER 47
STATICMEMBEROPT 47
STRING 8, 38
STRINGOPT 38
SYSTEMFRAME 46
TOKEN 26
UNDEFINED 37
VARIABLE 47
VARIABLETYPE 47
VARIABLEVALUE 47

A.4 Globals

add 98
addStaticBindings 59
assignArguments 145
attributeClass 150
badConstruct 76
bitAnd 105
bitNot 94
bitOr 106
bitwiseAnd 7
bitwiseOr 7
bitwiseShift 7
bitwiseXor 7
bitXor 106
booleanClass 149
bracketDelete 57
bracketRead 57
bracketWrite 57
call 89
characterClass 149
characterToCode 7
checkInteger 50
classClass 150
codeToCharacter 7
combineAttributes 56
construct 89
defineHoistedVar 61
defineInstanceMember 63
defineStaticMember 60
deleteDynamicProperty 76
deleteInstanceMember 75
deleteProperty 75
deleteReference 57
deleteStaticMember 75
divide 96
findFlatMember 66
findInstanceMember 68
findSlot 58
findStaticMember 67
findThis 65
float32Negate 12
float32ToFloat64 13
float32ToString 54
float64Abs 13
float64Add 14
float64Divide 15
float64Multiply 14
float64Negate 14
float64Remainder 15
float64Subtract 14
float64ToString 55
floatClass 149
functionClass 150
generalNumberClass 149
generalNumberCompare 51
generalNumberNegate 94
getEnclosingClass 58
getPackageOrGlobalFrame 58
getRegionalEnvironment 58
getRegionalFrame 58
getVariableType 74
hasType 52
instanceBindingsWithAccess 59
instantiateFrame 64
instantiateMember 64
instantiateOpenInstance 63
integerToLong 50
integerToString 53
integerToStringWithSign 53
integerToULong 50
isEqual 103
isLess 102
isLessOrEqual 102

isStrictEqual 104
lexicalDelete 65
lexicalRead 65
lexicalWrite 65
logicalNot 94
longClass 149
makeBuiltInClass 149
minus 94
multiply 96
namespaceClass 149
nullClass 149
numberClass 149
objectClass 149
objectPrototype 150
objectType 52
packageClass 150
plus 94
processPragma 135
prototypeClass 150
rationalToLong 50
rationalToULong 50
readDynamicProperty 70
readInstanceMember 69
readLimitedReference 84
readProperty 69
readReference 56
readStaticMember 70
readVariable 71
realToFloat32 11
realToFloat64 12
referenceBase 89
relaxedHasType 52
remainder 96
resolveAlias 51
resolveInstanceMemberName 68
resolveOverrides 62
searchForOverrides 62
selectPublicName 66
shiftLeft 99
shiftRight 100
shiftRightUnsigned 100
signedWrap32 49
signedWrap64 49
staticBindingsWithAccess 59
stringClass 149
subtract 98
toBoolean 52
toCompoundAttribute 56
toFloat64 51
toGeneralNumber 53
toPrimitive 55
toRational 50
toString 53
truncateFiniteFloat32 11
truncateFiniteFloat64 13
truncateToInteger 49
uLongClass 149
undefinedClass 149
unsignedWrap32 49
unsignedWrap64 49
writeDynamicProperty 74
writeInstanceMember 73
writeProperty 72
writeReference 57
writeStaticMember 73
writeVariable 74