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.................................. 4
   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
   5.7.1 Character Conversions........................... 8
   5.8 Lists.............................................. 8
   5.9 Strings........................................... 9
   5.10 Tuples.......................................... 10
   5.10.1 Shorthand Notation............................ 10
   5.11 Records......................................... 10
   5.12 ECMAScript Numeric Types......................... 11
   5.12.1 Signed Long Integers.......................... 11
       5.12.1.1 Shorthand Notation....................... 11
   5.12.2 Unsigned Long Integers........................ 12
       5.12.2.1 Shorthand Notation....................... 12
   5.12.3 Single-Precision Floating-Point Numbers........... 12
       5.12.3.1 Shorthand Notation....................... 12
   5.12.4 Double-Precision Floating-Point Numbers........... 13
       5.12.4.1 Shorthand Notation....................... 14
       5.12.4.2 Conversion................................ 14
       5.12.4.3 Arithmetic................................ 15
   5.13 Procedures ...................................... 16
   5.13.1 Operations................................... 17
   5.13.2 Semantic Domains of Procedures................... 17
   5.13.3 Steps........................................ 17
   5.13.4 Nested Procedures............................. 19
   5.14 Grammars......................................... 19
   5.14.1 Grammar Notation............................... 19
   5.14.2 Lookahead Constraints.......................... 20
   5.14.3 Line Break Constraints......................... 20
   5.14.4 Parameterised Rules........................... 20
   5.14.5 Special Lexical Rules.......................... 21
   5.15 Semantic Actions................................ 21
   5.15.1 Example ...................................... 22
   5.15.2 Abbreviated Actions........................... 23
   5.15.3 Action Notation Summary......................... 24
   5.16 Other Semantic Definitions........................ 26
6 Source Text........................................... 26
   6.1 Unicode Format-Control Characters.................. 26
7 Lexical Grammar....................................... 27
   7.1 Input Elements................................... 28
   7.2 White space .................................... 29
   7.3 Line Breaks..................................... 29
   7.4 Comments........................................ 30
   7.5 Keywords and Identifiers........................... 30
   7.6 Punctuators...................................... 33
   7.7 Numeric literals.................................. 33
   7.8 String literals................................... 36
   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.......................................... 38
   9.1.1 Undefined..................................... 39
   9.1.2 Null.......................................... 39
   9.1.3 Booleans...................................... 39
   9.1.4 Numbers...................................... 39
   9.1.5 Strings........................................ 39
   9.1.6 Namespaces..................................... 39
       9.1.6.1 Qualified Names............................ 40
   9.1.7 Compound attributes............................ 40
   9.1.8 Classes........................................ 40
       9.1.9 Simple Instances............................. 42
       9.1.9.1 Slots....................................... 43
   9.1.10 Uninstantiated Functions......................... 43
   9.1.11 Method Closures................................ 43
   9.1.12 Dates.......................................... 44
   9.1.13 Regular Expressions........................... 44
   9.1.14 Packages...................................... 44
   9.2 Objects with Limits................................ 44
   9.3 References........................................ 45
   9.4 Phases of evaluation................................ 46
   9.5 Contexts.......................................... 46
   9.6 Labels............................................ 46
   9.7 Semantic Exceptions............................... 46
   9.8 Function Support.................................. 47
   9.9 Environments...................................... 47
       9.9.1 Function Parameter Frames...................... 48
       9.9.1.1 Parameters.................................. 48
       9.9.2 Local Frames.................................. 49
       9.9.3 With Frames................................... 49
   9.10 Environment Bindings............................... 49
       9.10.1 Static Bindings............................... 49
       9.10.2 Instance Bindings............................. 51
   9.11 Miscellaneous................................... 52
       9.11.1 Extended Integers and Rationals................ 52
       9.11.2 Order......................................... 52
       9.11.3 Hints......................................... 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Data Operations</td>
</tr>
<tr>
<td>10.1</td>
<td>Numeric Utilities</td>
</tr>
<tr>
<td>10.2</td>
<td>Character Utilities</td>
</tr>
<tr>
<td>10.3</td>
<td>Object Utilities</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Object Class Inquiries</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Object to Boolean Conversion</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Object to Primitive Conversion</td>
</tr>
<tr>
<td>10.3.4</td>
<td>Object to Number Conversions</td>
</tr>
<tr>
<td>10.3.5</td>
<td>Object to String Conversions</td>
</tr>
<tr>
<td>10.3.6</td>
<td>Object to Qualified Name Conversion</td>
</tr>
<tr>
<td>10.3.7</td>
<td>Object to Class Conversion</td>
</tr>
<tr>
<td>10.3.8</td>
<td>Object to Attribute Conversion</td>
</tr>
<tr>
<td>10.3.9</td>
<td>Implicit Coercions</td>
</tr>
<tr>
<td>10.3.10</td>
<td>Attributes</td>
</tr>
<tr>
<td>10.4</td>
<td>Access Utilities</td>
</tr>
<tr>
<td>10.5</td>
<td>Environmental Utilities</td>
</tr>
<tr>
<td>10.6</td>
<td>Property Lookup</td>
</tr>
<tr>
<td>10.7</td>
<td>Reading</td>
</tr>
<tr>
<td>10.8</td>
<td>Writing</td>
</tr>
<tr>
<td>10.9</td>
<td>Deleting</td>
</tr>
<tr>
<td>10.10</td>
<td>Enumerating</td>
</tr>
<tr>
<td>10.11</td>
<td>Calling Instances</td>
</tr>
<tr>
<td>10.12</td>
<td>Creating Instances</td>
</tr>
<tr>
<td>10.13</td>
<td>Adding Local Definitions</td>
</tr>
<tr>
<td>10.14</td>
<td>Adding Instance Definitions</td>
</tr>
<tr>
<td>10.15</td>
<td>Instantiation</td>
</tr>
<tr>
<td>10.16</td>
<td>Sealing</td>
</tr>
<tr>
<td>10.17</td>
<td>Standard Class Utilities</td>
</tr>
<tr>
<td>11</td>
<td>Expressions</td>
</tr>
<tr>
<td>11.1</td>
<td>Terminal Actions</td>
</tr>
<tr>
<td>11.2</td>
<td>Identifiers</td>
</tr>
<tr>
<td>11.3</td>
<td>Qualified Identifiers</td>
</tr>
<tr>
<td>11.4</td>
<td>Primary Expressions</td>
</tr>
<tr>
<td>11.5</td>
<td>Function Expressions</td>
</tr>
<tr>
<td>11.6</td>
<td>Object Literals</td>
</tr>
<tr>
<td>11.7</td>
<td>Array Literals</td>
</tr>
<tr>
<td>11.8</td>
<td>Super Expressions</td>
</tr>
<tr>
<td>11.9</td>
<td>Postfix Expressions</td>
</tr>
<tr>
<td>11.10</td>
<td>Property Operators</td>
</tr>
<tr>
<td>11.11</td>
<td>Unary Operators</td>
</tr>
<tr>
<td>11.12</td>
<td>Multiplicative Operators</td>
</tr>
<tr>
<td>11.13</td>
<td>Additive Operators</td>
</tr>
<tr>
<td>11.14</td>
<td>Bitwise Shift Operators</td>
</tr>
<tr>
<td>11.15</td>
<td>Relational Operators</td>
</tr>
<tr>
<td>11.16</td>
<td>Equality Operators</td>
</tr>
<tr>
<td>11.17</td>
<td>Binary Bitwise Operators</td>
</tr>
<tr>
<td>11.18</td>
<td>Binary Logical Operators</td>
</tr>
<tr>
<td>11.19</td>
<td>Conditional Operator</td>
</tr>
<tr>
<td>11.20</td>
<td>Assignment Operators</td>
</tr>
<tr>
<td>11.21</td>
<td>Comma Expressions</td>
</tr>
<tr>
<td>11.22</td>
<td>Type Expressions</td>
</tr>
<tr>
<td>12</td>
<td>Statements</td>
</tr>
<tr>
<td>12.1</td>
<td>Empty Statement</td>
</tr>
<tr>
<td>12.2</td>
<td>Expression Statement</td>
</tr>
<tr>
<td>12.3</td>
<td>Super Statement</td>
</tr>
<tr>
<td>12.4</td>
<td>Block Statement</td>
</tr>
<tr>
<td>12.5</td>
<td>Labeled Statements</td>
</tr>
<tr>
<td>12.6</td>
<td>If Statement</td>
</tr>
<tr>
<td>12.7</td>
<td>Switch Statement</td>
</tr>
<tr>
<td>12.8</td>
<td>Do-While Statement</td>
</tr>
<tr>
<td>12.9</td>
<td>While Statement</td>
</tr>
<tr>
<td>12.10</td>
<td>For Statements</td>
</tr>
<tr>
<td>12.11</td>
<td>With Statement</td>
</tr>
<tr>
<td>12.12</td>
<td>Continue and Break Statements</td>
</tr>
<tr>
<td>12.13</td>
<td>Return Statement</td>
</tr>
<tr>
<td>12.14</td>
<td>Throw Statement</td>
</tr>
<tr>
<td>12.15</td>
<td>Try Statement</td>
</tr>
<tr>
<td>13</td>
<td>Directives</td>
</tr>
<tr>
<td>13.1</td>
<td>Attributes</td>
</tr>
<tr>
<td>13.2</td>
<td>Use Directive</td>
</tr>
<tr>
<td>13.3</td>
<td>Import Directive</td>
</tr>
<tr>
<td>13.4</td>
<td>Pragma</td>
</tr>
<tr>
<td>14</td>
<td>Definitions</td>
</tr>
<tr>
<td>14.1</td>
<td>Variable Definition</td>
</tr>
<tr>
<td>14.2</td>
<td>Simple Variable Definition</td>
</tr>
<tr>
<td>14.3</td>
<td>Function Definition</td>
</tr>
<tr>
<td>14.4</td>
<td>Class Definition</td>
</tr>
<tr>
<td>14.5</td>
<td>Namespace Definition</td>
</tr>
<tr>
<td>15</td>
<td>Programs</td>
</tr>
<tr>
<td>15.1</td>
<td>Package Definition</td>
</tr>
<tr>
<td>16</td>
<td>Predefined Identifiers</td>
</tr>
<tr>
<td>16.1</td>
<td>Built-in Namespaces</td>
</tr>
<tr>
<td>16.2</td>
<td>Built-in Attributes</td>
</tr>
<tr>
<td>16.3</td>
<td>Built-in Functions</td>
</tr>
<tr>
<td>17</td>
<td>Built-in Classes</td>
</tr>
<tr>
<td>17.1</td>
<td>Object</td>
</tr>
<tr>
<td>17.2</td>
<td>Never</td>
</tr>
<tr>
<td>17.3</td>
<td>Void</td>
</tr>
<tr>
<td>17.4</td>
<td>Null</td>
</tr>
<tr>
<td>17.5</td>
<td>Boolean</td>
</tr>
<tr>
<td>17.6</td>
<td>GeneralNumber</td>
</tr>
<tr>
<td>17.7</td>
<td>long</td>
</tr>
<tr>
<td>17.8</td>
<td>ulong</td>
</tr>
<tr>
<td>17.9</td>
<td>float</td>
</tr>
<tr>
<td>17.10</td>
<td>Number</td>
</tr>
<tr>
<td>17.11</td>
<td>char</td>
</tr>
<tr>
<td>17.12</td>
<td>String</td>
</tr>
<tr>
<td>17.13</td>
<td>Array</td>
</tr>
<tr>
<td>17.14</td>
<td>Namespace</td>
</tr>
<tr>
<td>17.15</td>
<td>Attribute</td>
</tr>
<tr>
<td>17.16</td>
<td>Date</td>
</tr>
<tr>
<td>17.17</td>
<td>RegExp</td>
</tr>
<tr>
<td>17.18</td>
<td>Class</td>
</tr>
<tr>
<td>17.19</td>
<td>Function</td>
</tr>
<tr>
<td>17.19.1</td>
<td>PrototypeFunction</td>
</tr>
<tr>
<td>17.20</td>
<td>Package</td>
</tr>
<tr>
<td>17.21</td>
<td>Error</td>
</tr>
<tr>
<td>17.21.1</td>
<td>Error Subclasses</td>
</tr>
</tbody>
</table>

A Index

A.1 Nonterminals
A.2 Tags
A.3 Semantic Domains
A.4 Globals
1 Scope

This Standard defines the ECMAScript Edition 4 scripting language.

2 Conformance

3 Normative References

4 Overview

5 Notational Conventions

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

5.1 Text

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

When denoted in this specification, characters with values code points between 20 and 7E hexadecimal inclusive are in a fixed width font. Other characters are denoted by enclosing their four-digit hexadecimal Unicode code points value between «u» and ». Supplementary Unicode characters (code points from 10000 to 10FFFF hexadecimal) are denoted by enclosing their eight-digit hexadecimal Unicode code points between «U» and ». For example, the non-breakable space character would be denoted in this document as «u00A0», and the character with the code point 1234F hexadecimal would be denoted as «U0001234F». A few of the common control characters are represented by name:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>«NUL»</td>
<td>«u0000»</td>
</tr>
<tr>
<td>«BS»</td>
<td>«u0008»</td>
</tr>
<tr>
<td>«TAB»</td>
<td>«u0009»</td>
</tr>
<tr>
<td>«LF»</td>
<td>«u000A»</td>
</tr>
<tr>
<td>«VT»</td>
<td>«u000B»</td>
</tr>
<tr>
<td>«FF»</td>
<td>«u000C»</td>
</tr>
<tr>
<td>«CR»</td>
<td>«u000D»</td>
</tr>
<tr>
<td>«SP»</td>
<td>«u0020»</td>
</tr>
</tbody>
</table>

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

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

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

Semantic domains have names in CAPITALISED SMALL CAPS. Such a name is to be considered distinct from a tag or regular variable with the same name, so UNDEFINED, undefined, and undefined are three different and independent entities.

A variable \(v\) is constrained using the notation

\[ v : T \]

where \(T\) is a semantic domain. This constraint indicates that the value of \(v\) will always be a member of the semantic domain \(T\). These declarations are informative (they may be dropped without affecting the semantics’ correctness) but useful in understanding the semantics. For example, when the semantics state that \(x: INTEGER\) then one does not have to worry about what happens when \(x\) has the value true or \(+\infty\).

The constraints can be proven statically. The semantics have been machine-checked to ensure that every constraint holds.

5.3 Tags

Tags are computational tokens with no internal structure. Tags are written using a bold sans-serif font. Two tags are equal if and only if they have the same name. Examples of tags include true, false, null, NaN, and identifier.

5.4 Booleans

The tags true and false represent Booleans. BOOLEAN is the two-element semantic domain \{true, false\}.

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

\[
\begin{align*}
\text{not } a & \quad \text{true if } a \text{ is false; false if } a \text{ is true} \\
\text{and } a \text{ and } b & \quad \text{if } a \text{ is false}, \text{returns false without computing } b; \text{ if } a \text{ is true}, \text{returns the value of } b \\
\text{or } a \text{ or } b & \quad \text{if } a \text{ is false}, \text{returns the value of } b; \text{ if } a \text{ is true}, \text{returns true without computing } b \\
xor a \text{ xor } b & \quad \text{true if } a \text{ is true and } b \text{ is false or } a \text{ is false and } b \text{ is true; false otherwise. } a \text{ xor } b \text{ is equivalent to } a \neq b
\end{align*}
\]

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

5.5 Sets

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

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

\[ \{element_1, element_2, \ldots, element_n\} \]

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

For example, the set \{3, 0, 10, 11, 12, 13, -5\} contains seven integers.
Sets of either integers or characters can be abbreviated using the ... range operator, which generates inclusive ranges of integers or character code points. For example, the above set can also be written as \{0, \ldots, 3, 10 \ldots, 13\}.

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

A set can also be written using the set comprehension notation

\{f(x) \mid 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 \text{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 such that } x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}
\{x \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
\(x \notin A\) false if \(x\) is an element of \(A\) and true if not
\(|A|\) The number of elements in \(A\) (only used on finite sets)
\(\min A\) The value \(m\) that satisfies both \(m \notin A\) 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 \(m \notin A\) 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 \(B\))
\(A \setminus B\) The difference of \(A\) and \(B\) (the set or semantic domain of all values that are present in \(A\) but not \(B\))
\(A = B\) true if \(A\) and \(B\) are equal and false otherwise. \(A\) and \(B\) are equal if every element of \(A\) is also in \(B\) and every element of \(B\) is also in \(A\).
\(A \neq B\) false if \(A\) and \(B\) are equal and true otherwise
\(A \subseteq B\) true if \(A\) is a subset of \(B\) and false otherwise. \(A\) is a subset of \(B\) if every element of \(A\) is also in \(B\). Every set is a subset of itself. The empty set \(\{\}\) is a subset of every set.
\(A \supset B\) true if \(A\) is a proper subset of \(B\) and false otherwise. \(A \supset B\) is equivalent to \(A \subseteq B\) and \(A \neq B\).

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

\(T = \{1,2,3\}\)

then:

\(T\{\}\) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}

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

In addition to the above, the \text{some} and \text{every} quantifiers can be used on sets. The quantifier

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

returns \text{true} if there exists at least one element \(x\) in set \(A\) such that \text{predicate}(x) computes to \text{true}. If there is no such element \(x\), then the \text{some} quantifier’s result is \text{false}. If the \text{some} quantifier returns \text{true}, then variable \(x\) is left bound to any element of \(A\) for which \text{predicate}(x) computes to \text{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:

```
(some x [] {3, 16, 19, 26} satisfies x mod 10 = 7) = false;
(some x [] {1} satisfies x mod 10 = 7) = false;
(some x [] {"Hello"}) satisfies true) = true and leaves `x` set to the string “Hello”;
(some x [] {}) satisfies true) = false.
```

The quantifier

```
every x [] A satisfies predicate(x)
```

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

```
(every x [] {3, 16, 19, 26} satisfies x mod 10 = 6) = false;
(every x [] {6, 26, 106} satisfies x mod 10 = 6) = true;
(every x [] {}) satisfies x mod 10 = 6) = true.
```

### 5.6 Real Numbers

Numbers written in this specification are to be understood to be exact mathematical real numbers, which include integers and rational numbers as subsets. Examples of numbers include -3, 0, 17, 10^{100}, and[]. Hexadecimal numbers are written by preceding them with “0x”, so 4294967296, 0x100000000, and 2^{32} are all the same integer.

**INTEGER** is the semantic domain of all integers {... –3, –2, –1, 0, 1, 2, 3 ...}. 3.0, 3, 0xFF, and –10^{100} are all integers.

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

**REAL** is the semantic domain of all real numbers. Every rational number is also a real number: RATIONAL [] REAL. [] 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 [] y` Product
- `x / y` Quotient (`y` must not be zero)
- `x^y` `x` raised to the `y`th power (used only when either `x`≠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`≥0 and `–x` otherwise
- `floored x` Floor of `x`, which is the unique integer `i` such that `i`≤`x`<`i`+1. `floored 3.5` = 3, `floored -3.5` = –4, and `floor 7` = 7.
- `ceiled x` Ceiling of `x`, which is the unique integer `i` such that `i`–1<`x`≤`i`. `ceiled 7` = 4, `ceiled -3.5` = –3, and `ceiled 7` = 7.
- `x mod y` `x` modulo `y`, which is defined as `x – y {
\[\left\lfloor\frac{x}{y}\right\rfloor\]} y` must not be zero. `10 mod 7` = 3, and `–1 mod 7` = 6.
- `log_{10}(x)` The exact base-10 logarithm of `x` (this will always be greater than zero)

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

### 5.6.1 Bitwise Integer Operators

The four procedures below perform bitwise operations on integers. The integers are treated as though they were written in infinite-precision two’s complement binary notation, with each 1 bit representing `true` and 0 bit representing `false`. 
More precisely, any integer \( x \) can be represented as an infinite sequence of bits \( a_i \) where the index \( i \) ranges over the nonnegative integers and every \( a_i \in \{0, 1\} \). The sequence is traditionally written in reverse order:

\[
..., a_3, a_2, a_1, a_0
\]

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

\[
a_i = \lfloor x / 2^i \rfloor \mod 2
\]

If \( x \) is zero or positive, then its sequence will have infinitely many consecutive leading 0’s, while a negative integer \( x \) will generate a sequence with infinitely many consecutive leading 1’s. For example, 6 generates the sequence \( ...0...0000110 \), while \(-6\) generates \( ...1...111010 \).

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 \( ...0...0000110 \), which is the sequence generated by the integer 2. Thus, bitwiseAnd(6, \(-6\)) = 2.

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

### 5.7 Characters

Characters enclosed in single quotes ‘ and ’ represent single Unicode characters with code points ranging from 0000 to 10FFFF hexadecimal. Even though Unicode does not define characters for some of these code points, in this specification any of these 1114112 code points is considered to be a valid character. Examples of characters include ‘A’, ‘b’, ‘\(10000\)’, and ‘\(uFFFF\)’; ‘\(u00010000\)’ and ‘\(u00010FFFF\)’ (see also section 5.1). Unicode surrogates are considered to be pairs of characters for the purpose of this specification.

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

CHAR16 is the semantic domain of the 65536 Unicode characters in the setall 65536 characters {‘\(u0000\)’ ... ‘\(uFFFF\)’}. These characters form Unicode’s Basic Multilingual Plane. These characters have code points between 0000 and FFFF hexadecimal. Code units are also represented by values in the CHAR16 semantic domain.

SUPPLEMENTARYCHAR is the semantic domain of the 1048576 Unicode characters in the set {‘\(u00010000\)’ ... ‘\(u0010FFFF\)’}. These are Unicode’s supplementary characters with code points between 10000 and 10FFFF hexadecimal. Since these characters are not members of the CHAR16 domain, they cannot be stored directly in strings of CHAR16 code units. Instead, wherever necessary the semantic algorithms convert supplementary characters into pairs of surrogate code units before storing them into strings. The first surrogate code unit \( h \) is in the set {‘\(u0800\)’ ... ‘\(uDBFF\)’} and the second surrogate code unit \( l \) is in the set {‘\(uDC00\)’ ... ‘\(uDFFF\)’}; together they encode the supplementary character with the code point value:

\[
0x10000 + (16ToInteger(h) – 0xD800) \cdot 0x400 + 16ToInteger(l) – 0xDC00
\]

CHAR21 is the semantic domain of all 1114112 Unicode characters {‘\(u0000\)’ ... ‘\(u0010FFFF\)’}:

\[
CHAR21 = CHAR16 \cup SUPPLEMENTARYCHAR
\]

Characters can be compared using \( = \), \( \neq \), \( < \), \( \leq \), \( > \), and \( \geq \). These operators compare code point values, so ‘\(A\)’ = ‘\(A\)’, ‘\(A\)’ < ‘\(B\)’, and ‘\(A\)’ < ‘\(a\)’, and ‘\(uFFFF\)’ < ‘\(u00010000\)’ are all true.
5.7.1 Character Conversions

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

- `char16ToInteger(c: CHAR16): {0 ... 0xFFFF}` characterToCode(c: CHARACTER): {0 ... 65535}
  
  Return the number of character `c`’s Unicode code point or code unit. Return character `c`’s Unicode code point as an integer.

- `char21ToInteger(c: CHAR21): {0 ... 0x10FFFF}` codeToCharacter(i: {0 ... 65535}): CHARACTER
  
  Return the number of character `c`’s Unicode code point. Return the character whose Unicode code point is `i`.

- `integerToChar16(i: {0 ... 0xFFFF}): CHAR16`
  
  Return the character whose Unicode code point number is `i`.

- `integerToSupplementaryChar(i: {0x10000 ... 0x10FFFF}): SUPPLEMENTARYCHAR`
  
  Return the character whose Unicode code point number is `i`.

- `integerToChar21(i: {0 ... 0x10FFFF}): CHAR21`
  
  Return the character whose Unicode code point number is `i`.

The procedure `digitValue` is defined as follows:

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

5.8 Lists

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

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

For example, the following list contains four strings:

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

The empty list is written as `[]`.

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

A list can also be written using the list comprehension notation

\[
[f(x) | [x ∈ u]]
\]

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

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

denotes the list of the results of computing expression \(f\) on all elements \(x\) of list \(u\) that satisfy the `predicate` expression. The results are listed in the same order as the elements \(x\) of list \(u\). For example,
Let \( u = [e_0, e_1, \ldots, e_n] \) and \( v = [f_0, f_1, \ldots, f_m] \) 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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Precondition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>u</td>
<td>)</td>
</tr>
<tr>
<td>( u[i] )</td>
<td>( 0 \leq i &lt;</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \ldots j] )</td>
<td>( 0 \leq i \leq j + 1 \leq</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \ldots] )</td>
<td>( 0 \leq i \leq</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \setminus x] )</td>
<td>( 0 \leq i &lt;</td>
<td>u</td>
</tr>
<tr>
<td>( u \oplus v )</td>
<td>( 0 \leq j &lt;</td>
<td>u</td>
</tr>
<tr>
<td>( \text{repeat}(e, i) )</td>
<td>( i \geq 0 )</td>
<td>The list ( [e, e, \ldots, e] ) of length ( i ) containing ( i ) identical elements ( e )</td>
</tr>
<tr>
<td>( u = v )</td>
<td>( \text{true} ) if the lists ( u ) and ( v ) are equal and ( \text{false} ) otherwise. Lists ( u ) and ( v ) are equal if they have the same length and all of their corresponding elements are equal.</td>
<td></td>
</tr>
<tr>
<td>( u \neq v )</td>
<td>( \text{false} ) if the lists ( u ) and ( v ) are equal and ( \text{true} ) otherwise.</td>
<td></td>
</tr>
</tbody>
</table>

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 \textbf{some} and \textbf{every} quantifiers can be used on lists just as on sets:

\begin{verbatim}
  some \( x \in u \) satisfies \textit{predicate} \( x \)
  every \( x \in u \) satisfies \textit{predicate} \( x \)
\end{verbatim}

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

\begin{verbatim}
  some \( x \in [3, 36, 19, 26] \) satisfies \textit{x mod 10 = 6}
\end{verbatim}

evaluates to \textbf{true} and leaves \( x \) set to 36.

### 5.9 Strings

A list of \texttt{CHAR16 code unit} characters is called a \textit{string}. In addition to the normal list notation, for notational convenience a string can also be written as zero or more characters enclosed in double quotes (see also section 5.1 the notation for non-ASCII characters). Thus,

\begin{verbatim}
  “Wonder\textbackslash r\textbackslash n”
\end{verbatim}

is equivalent to:

\begin{verbatim}
  [‘W’, ‘\r’, ‘n’, ‘\d’, ‘\e’, ‘x’, ‘\r\n’]
\end{verbatim}

The empty string is usually written as \texttt{“”}.

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

In addition to the other list operations, \texttt{<}, \texttt{\leq}, \texttt{>}, and \texttt{\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 \texttt{code unit} character of \( x \) is less than the first \texttt{code unit} character of \( y \), or the first \texttt{code unit} character of \( x \) is equal to the first \texttt{code unit} character of \( y \) and the rest of string \( x \) is less than the rest of string \( y \).
Note that these relations compare code units, not code points, which can produce unexpected effects if a string contains supplementary characters expanded into a pair of surrogates. For example, even though ‘«uFFFF»’ < ‘«U00010000»’, the supplementary character ‘«U00010000»’ is represented in a string as “«uD800»«uDC00»”, and, by the above rules, “«uFFFF»” > “«uD800»«uDC00»”.

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

5.10 Tuples

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>label1</td>
<td>T1</td>
<td>Informative note about this field</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>labeln</td>
<td>Tn</td>
<td>Informative note about this field</td>
</tr>
</tbody>
</table>

label1 through labeln are the names of the fields. T1 through Tn are informative semantic domains of possible values that the corresponding fields may hold.

The notation

```
NAME[label1: v1, ..., labeln: vn]
```

represents a tuple with name NAME and values v1 through vn for fields labelled label1 through labeln respectively. Each value vi is a member of the corresponding semantic domain Ti. When most of the fields are copied from an existing tuple a, this notation can be abbreviated as

```
NAME[label1: v1, ..., labeln: vn, other fields from a]
```

which represents a tuple with name NAME and values v1 through vn for fields labelled label1 through labeln respectively and the values of correspondingly labeled fields from a for all other fields.

If a is the tuple NAME[label1: v1, ..., labeln: vn] then

```
a.labeli
```

returns the i<sup>th</sup> field’s value vi.

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

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

5.10.1 Shorthand Notation

The semantic notation ns::id is a shorthand for QUALIFIEDNAME[namespace: ns, id: id] See section 9.1.6.1.

5.11 Records

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

A record is comprised of a name NAME and an *address*. The address points to a mutable data structure comprised of zero or more labelled fields. The address acts as the record’s serial number — every record allocated by new (see below) gets a different address, including records created by identical expressions or even the same expression used twice.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>label1</td>
<td>T1</td>
<td>Informative note about this field</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
The expression

\[
\text{new NAME} [\text{label}_1, \ldots, \text{label}_n] \quad \text{new expression for fields} \]

creates a record with name NAME and a new address [ ]. The fields labelled \text{label}_1 through \text{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 \(\text{label}_i: v_i\) pair may be omitted from a new expression, which indicates that the initial value of field \text{label}_i 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

\[
\text{new NAME} [\text{label}_1, \ldots, \text{label}_n]\quad \text{new expression for fields} \]

which represents a record \(b\) with name NAME and a new address [ ]. The fields labelled \text{label}_j through \text{label}_k at address [ ] are initialised with values \(v_{i_j}\) through \(v_{i_k}\) respectively; the other fields at address [ ] are initialised with the values of correspondingly labeled fields from \(a\)'s address.

If \(a\) is a record with name NAME and address [ ], then

\[
\text{a.\text{label}_i} \quad \text{return current value of field} \]

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

\[
\text{a.\text{label}_i}[w] \quad \text{set value of field} \]

after which \(\text{a.\text{label}_i}\) will evaluate to \(w\). Any record with a different address [ ] 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 \texttt{GENERALNUMBER}, defined as the union of four basic numeric semantic domains \texttt{LONG}, \texttt{ULONG}, \texttt{FLOAT32}, and \texttt{FLOAT64}:

\[
\texttt{GENERALNUMBER} = \texttt{LONG} \quad \texttt{ULONG} \quad \texttt{FLOAT32} \quad \texttt{FLOAT64}
\]

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

The semantic domain \texttt{FINITEGENERALNUMBER} is the subtype of all finite values in \texttt{GENERALNUMBER}:

\[
\texttt{FINITEGENERALNUMBER} = \texttt{LONG} \quad \texttt{ULONG} \quad \texttt{FLOAT32} \quad \texttt{FLOAT64}
\]

### 5.12.1 Signed Long Integers

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>{-2^{63} ... 2^{63} - 1}</td>
<td>The signed 64-bit integer</td>
</tr>
</tbody>
</table>

### 5.12.1.1 Shorthand Notation

In this specification, when \(i\) is an integer between \(-2^{63}\) and \(2^{63} - 1\), the notation \(i_{\text{long}}\) indicates the result of \texttt{LONG} \[\text{value: } i\] which is the integer \(i\) wrapped in a \texttt{LONG} tuple.
5.12.2 Unsigned Long Integers

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>{0 ... 2^{64} – 1}</td>
<td>The unsigned 64-bit integer</td>
</tr>
</tbody>
</table>

5.12.2.1 Shorthand Notation

In this specification, when \(i\) is an integer between 0 and \(2^{64} – 1\), the notation \(i\text{\(_{\text{ulong}}\)}\) indicates the result of ULONG\[\text{value} = i\] which is the integer \(i\) wrapped in a ULONG tuple.

5.12.3 Single-Precision Floating-Point Numbers

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

\[
\text{FLOAT32} = \text{FINITEFLOAT32} \uplus \{+\text{inf}_{\text{f32}}, –\text{inf}_{\text{f32}}, \text{NaN}_{\text{f32}}\}; \\
\text{FINITEFLOAT32} = \text{NONZEROFINITEFLOAT32} \uplus \{+0_{\text{f32}}, –0_{\text{f32}}\}
\]

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>NORMALISEDFLOAT32VALUES \uplus DENORMALISEDFLOAT32VALUES</td>
<td>The value, represented as an exact rational number</td>
</tr>
</tbody>
</table>

There are 4294967296 (that is, \(2^{32} – 2^{25}\)) normalised values:

\[
\text{NORMALISEDFLOAT32VALUES} = \{s\, m\, \underbrace{2^{23}}_{s\, e} \mid s \in \{\pm 1\}, m \in \{2^{23} \ldots 2^{24} – 1\}, e \in \{-149 \ldots 104\}\}
\]

\(m\) is called the significand.

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

\[
\text{DENORMALISEDFLOAT32VALUES} = \{s\, m\, \underbrace{2^{23}}_{s\, e} \mid s \in \{\pm 1\}, m \in \{1 \ldots 2^{23} – 1\}\}
\]

\(m\) is called the significand.

The remaining FLOAT32 values are the tags +0\(_{\text{f32}}\) (positive zero), –0\(_{\text{f32}}\) (negative zero), +\text{inf}_{\text{f32}}\ (positive infinity), –\text{inf}_{\text{f32}}\ (negative infinity), and NaN\(_{\text{f32}}\) (not a number).

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

Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation = and \# may be used to compare them. Note that = is false for different tags, so +0\(_{\text{f32}}\) \# –0\(_{\text{f32}}\) but NaN\(_{\text{f32}}\) = NaN\(_{\text{f32}}\). 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 \(x_{\text{f32}}\) indicates the result of \(\text{realToFloat32}(x)\), which is the “closest” FLOAT32 value as defined below. Thus, 3.4 is a REAL number, while 3.4\(_{\text{f32}}\) is a FLOAT32 value (whose exact value is actually 3.400000095367431640625). The positive finite FLOAT32 values range from \(10^{-125}\) to \(3.4028235 \times 10^{38}\) \(_{\text{f32}}\).

5.12.3.2 Conversion

The procedure \(\text{realToFloat32}\) converts a real number \(x\) into the applicable element of FLOAT32 as follows:
There are 18428729675200069632 (that is, 2\(^{53}\)) domains:

The values considered indistinguishable from each other.

\[ 5.12.4 \]

The procedure \texttt{realToFloat32} takes a \texttt{RATIONAL} value and converts it to a \texttt{REAL32} value:

\begin{verbatim}
proc realToFloat32(x: REAL): REAL32
    s: RATIONAL
    NORMALISEDREAL32VALUES
    DENORMALISEDREAL32VALUES { -2\(^{128}\), 0, 2\(^{128}\)};
    Let \(a\) be \texttt{RATIONAL} be the element of \(s\) closest to \(x\) (i.e. such that \(|a-x|\) is as small as possible). If two elements of \(s\) are equally close, let \(a\) be the one with an even significand; for this purpose \(-2\(^{128}\), 0, and 2\(^{128}\) are considered to have even significands.
    if \(a = 2\(^{128}\) then return \(+\infty\)32
    elsif \(a = -2\(^{128}\) then return \(-\infty\)32
    elsif \(a \neq 0\) then return NONZEROREAL32\{value: \(a\}\}
    elsif \(x < 0\) then return \(-\texttt{ZERO}32\)
    else return \(+\texttt{ZERO}32\)
end if
end proc
\end{verbatim}

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

The procedure \texttt{truncateFiniteFloat32} truncates a \texttt{REAL32} value to an integer, rounding towards zero:

\begin{verbatim}
proc truncateFiniteFloat32(x: REAL32): INTEGER
    if \(x \in \{0, \texttt{ZERO}32\}\) then return 0 end if;
    r: RATIONAL x value;
    if \(r > 0\) then return \(\lceil x \rceil\) else return \(\lfloor x \rfloor\) end if
end proc
\end{verbatim}

\subsection*{5.12.3.3 Arithmetic}

The following table defines negation of \texttt{REAL32} values using IEEE 754 rules. Note that \((\texttt{expr})_{32}\) is a shorthand for \texttt{realToFloat32}((\texttt{expr})).

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{x} & \textbf{Result} \\
\hline
\(\texttt{ZERO}32\) & \(+\texttt{ZERO}32\) \\
\(\texttt{ZERO}32\) & \(+\texttt{ZERO}32\) \\
\(+\texttt{ZERO}32\) & \(-\texttt{ZERO}32\) \\
\(-\texttt{ZERO}32\) & \(-\texttt{ZERO}32\) \\
\(+\infty_{32}\) & \(-\infty_{32}\) \\
\(-\infty_{32}\) & \(+\infty_{32}\) \\
\hline
\end{tabular}
\end{table}

\section*{5.12.4 Double-Precision Floating-Point Numbers}

\texttt{REAL64} 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. \texttt{REAL64} is the union of the following semantic domains:

\[ \text{REAL64} = \text{REAL32} \uplus \{ \pm \infty, \texttt{ZERO}64, \texttt{NaN}64 \}; \]

\[ \text{REAL32} = \{ \pm \infty, \texttt{ZERO}32, \texttt{NaN}32 \}; \]

The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains \texttt{REAL32}, \texttt{REAL64}, \texttt{LONG}, \texttt{ULONG}, \texttt{FLOAT32}, \texttt{FLOAT64}, \texttt{NORMALISEDREAL32VALUES}, \texttt{DENORMALISEDREAL32VALUES}, \texttt{NORMALISEDREAL64VALUES}, \texttt{DENORMALISEDREAL64VALUES}

\begin{verbatim}
Field         Contents                      Note
\hline
value         NORMALISEDREAL64VALUES up DENORMALISEDREAL64VALUES  \\
\hline
\end{verbatim}

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

\[ \text{NORMALISEDREAL64VALUES} = \{ s \uparrow m \uparrow 2^c \uparrow s \uparrow \{-1, 1\}, \uparrow m \uparrow \{2^{52} \ldots 2^{51}\}, \uparrow c \uparrow \{-1074 \ldots 971\} \} \]

\(m\) is called the significand.
There are also 9007199254740990 (that is, $2^{53} - 2$) denormalised non-zero values:

\[ \text{DENORMALISEDFLOAT64VALUES} = \{ s \in \{-1, 1\} \mid s \neq 0 \\land s \not\in \{-1, 1\} \} \]

\( m \) is called the significand.

The remaining FLOAT64 values are the tags +zero\(_{64}\) (positive zero), –zero\(_{64}\) (negative zero), +\( \infty \)\(_{64}\) (positive infinity), –\( \infty \)\(_{64}\) (negative infinity), and NaN\(_{64}\) (not a number).

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

Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation = and ≠ may be used to compare them. Note that = is false for different tags, so +zero\(_{64}\) ≠ –zero\(_{64}\) but NaN\(_{64}\) = NaN\(_{64}\). 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 \( x\)\(_{64}\) indicates the result of realToFloat64(\( x \)), which is the “closest” FLOAT64 value as defined below. Thus, \( 3.4 \) is a REAL number, while \( 3.4\)\(_{64}\) is a FLOAT64 value (whose exact value is actually \( 3.399999999999999991182158029987476766109466552734375 \)). The positive finite FLOAT64 values range from \( 5 \mid 10^{-32} \)\(_{64}\) to \( (1.7976931348623157 \mid 10^{308})\)\(_{64}\).

### 5.12.4.2 Conversion

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

\[
\text{proc realToFloat64} (x:\ \text{REAL}) : \text{FLOAT64} \\
\text{x} : \text{RATIONAL} ; \text{NONORMALISEDFLOAT64VALUES} \text{DENORMALISEDFLOAT64VALUES} \text{\{} 2^{1024}, 0, 2^{1024} \text{\};} \\
\text{Let } a : \text{RATIONAL} \text{ be the element of } s \text{ closest to } x \text{ (i.e. such that } |a-x| \text{ is as small as possible). If two elements of } s \text{ are equally close, let } a \text{ be the one with an even significand; for this purpose } 2^{1024}, 0, \text{ and } 2^{1024} \text{ are considered to have even significands.} \\
\text{if } a = 2^{1024} \text{ then return } +\infty \text{\(_{64}\);} \\
\text{elseif } a = -2^{1024} \text{ then return } -\infty \text{\(_{64}\);} \\
\text{elseif } a \neq 0 \text{ then return NONZEROFINITEFLOAT64\{value: } a\}; \\
\text{else return } +\text{zero}\text{\(_{64}\);} \\
\text{end return;}
\]

**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:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( = \text{FLOAT32} )</th>
<th>( \text{FLOAT64} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( -\infty )(_{32})</td>
<td>( -\infty )(_{64})</td>
<td></td>
</tr>
<tr>
<td>( -\text{zero})(_{32})</td>
<td>( -\text{zero})(_{64})</td>
<td></td>
</tr>
<tr>
<td>( +\text{zero})(_{32})</td>
<td>( +\text{zero})(_{64})</td>
<td></td>
</tr>
<tr>
<td>( +\infty )(_{32})</td>
<td>( +\infty )(_{64})</td>
<td></td>
</tr>
<tr>
<td>NaN(_{32})</td>
<td>NaN(_{64})</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{proc truncateFiniteFloat64} (x:\ \text{FINITEFLOAT64}) : \text{INTEGER} \\
\text{if } x \text{\{+zero\}\(_{64}\) –zero\(_{64}\}) \text{ then return 0 end if;} \\
\text{r} : \text{RATIONAL} ; x\text{.value} ; \\
\text{if } r > 0 \text{ then return } x\text{.value else return } x\text{.value} \text{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

<table>
<thead>
<tr>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+\infty_{f64}</td>
<td>+\infty_{f64}</td>
</tr>
<tr>
<td>negative finite</td>
<td>(-x.value)_{f64}</td>
</tr>
<tr>
<td>-zero_{f64}</td>
<td>+zero_{f64}</td>
</tr>
<tr>
<td>+zero_{f64}</td>
<td>+zero_{f64}</td>
</tr>
<tr>
<td>positive finite</td>
<td>x</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td>+\infty_{f64}</td>
</tr>
<tr>
<td>NaN_{f64}</td>
<td>NaN_{f64}</td>
</tr>
</tbody>
</table>

#### float64Negate(x: FLOAT64): FLOAT64

<table>
<thead>
<tr>
<th>x</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+\infty_{f64}</td>
<td>+\infty_{f64}</td>
</tr>
<tr>
<td>negative finite</td>
<td>(-x.value)_{f64}</td>
</tr>
<tr>
<td>-zero_{f64}</td>
<td>+zero_{f64}</td>
</tr>
<tr>
<td>+zero_{f64}</td>
<td>-zero_{f64}</td>
</tr>
<tr>
<td>positive finite</td>
<td>(-x.value)_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td>-\infty_{f64}</td>
</tr>
<tr>
<td>NaN_{f64}</td>
<td>NaN_{f64}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>+\infty_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>+\infty_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
<tr>
<td>+\infty_{f64}</td>
<td></td>
<td>NaN_{f64}</td>
</tr>
</tbody>
</table>

**NOTE** The identity for floating-point addition is \(-zero_{f64}\), not +zero_{f64}.
A procedure is denoted as:

```
proc (param1: T1, ... , paramn: Tn): T
    step;
    step;
    ...;
    step;
end proc;
```

### 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.

**NOTE**  

`float64Remainder(float64Negate(x), y)` always produces the same result as `float64Negate(float64Remainder(x, y))`. Also, `float64Remainder(x, float64Negate(y))` always produces the same result as `float64Remainder(x, y)`.  

#### float64Multiply(x: Float64, y: Float64): Float64

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞f64</td>
<td>negative finite</td>
<td>-∞f64</td>
</tr>
<tr>
<td>+∞f64</td>
<td>negative finite</td>
<td>+∞f64</td>
</tr>
<tr>
<td>-zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>+zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>NaNf64</td>
<td>NaNf64</td>
<td>NaNf64</td>
</tr>
</tbody>
</table>

#### float64Divide(x: Float64, y: Float64): Float64

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞f64</td>
<td>negative finite</td>
<td>-∞f64</td>
</tr>
<tr>
<td>+∞f64</td>
<td>negative finite</td>
<td>+∞f64</td>
</tr>
<tr>
<td>-zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>+zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>NaNf64</td>
<td>NaNf64</td>
<td>NaNf64</td>
</tr>
</tbody>
</table>

#### float64Remainder(x: Float64, y: Float64): Float64

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞f64</td>
<td>+∞f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>-zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>+zero f64</td>
<td>+zero f64</td>
<td>NaNf64</td>
</tr>
<tr>
<td>NaNf64</td>
<td>NaNf64</td>
<td>NaNf64</td>
</tr>
</tbody>
</table>

### 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 (param1: T1, ... , paramn: Tn): T
    step;
    step;
    ...;
    step;
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 stepₙ 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.

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

eexpression
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 \text{expression}_1\text{ then }\text{step}; \text{step}; ...; \text{step} \\
elsif \text{expression}_2\text{ then }\text{step}; \text{step}; ...; \text{step} \\
... \\
elsif \text{expression}_n\text{ then }\text{step}; \text{step}; ...; \text{step} \\
else \text{step}; \text{step}; ...; \text{step} \\
end if

An if step computes \text{expression}_1, which will evaluate to either true or false. If it is true, the first list of steps is performed. Otherwise, \text{expression}_2 is computed and tested, and so on. If no \text{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 \text{expression} evaluates to true.

case \text{expression} of 
\text{T}_1\text{ do }\text{step}; \text{step}; ...; \text{step}; \\
\text{T}_2\text{ do }\text{step}; \text{step}; ...; \text{step}; \\
...; \\
\text{T}_n\text{ do }\text{step}; \text{step}; ...; \text{step} \\
else \text{step}; \text{step}; ...; \text{step} \\
end case

A case step computes \text{expression}, which will evaluate to a value \text{v}. If \text{v} \in \text{T}_1, then the first list of steps is performed. Otherwise, if \text{v} \not\in \text{T}_2, then the second list of steps is performed, and so on. If \text{v} is not a member of any \text{T}_i, the list of steps following the else is performed. The else clause may be omitted, in which case \text{v} will always be a member of some \text{T}_i.

while \text{expression} do 
\text{step}; \\
...; \\
\text{step} \\
end while

A while step computes \text{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 \text{expression} is computed and tested again. This repeats until \text{expression} returns true (or until the procedure exits via a return or an exception is propagated out).

for each \text{x} \in \text{expression} do 
\text{step}; \\
...; \\
\text{step} \\
end for each

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

return \text{expression}

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

invariant \text{expression}

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

throw \text{expression}

A throw step computes \text{expression} to obtain a value \text{v} and begins propagating exception \text{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

\[
SampleList []
\]

\[
| . . . Identifier \quad (Identifier: 11.2)
| SampleListPrefix
| SampleListPrefix, . . . Identifier
\]

states that the nonterminal \emph{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 \emph{...} followed by any expansion of the nonterminal \emph{Identifier}.
- It can represent any expansion of the nonterminal \emph{SampleListPrefix}.
- It can represent any expansion of the nonterminal \emph{SampleListPrefix} followed by the terminals \emph{,} and \emph{...} and any expansion of the nonterminal \emph{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 \emph{set}. That \emph{set} can be written as a list of terminals enclosed in curly braces. For convenience, \emph{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 \emph{n} followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

### 5.14.3 Line Break Constraints

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

\[
ReturnStatement[]
return
| return [no line break] ListExpression\textsuperscript{allowIn}
\]

indicates that the second production may not be used if a line break occurs in the program between the \emph{return} token and the \emph{ListExpression}\textsuperscript{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

\[
\emptyset \quad \{normal, initial\}
\]
introduce grammar arguments \([\allowIn, \noIn]\). 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

\[
\text{AssignmentExpression}{}^{[\allowIn \noIn]} \text{ ConditionalExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\allowIn]} = \text{AssignmentExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\allowIn]} \text{ CompoundAssignment AssignmentExpression}{}^{[\allowIn \noIn]}
\]

expands into the following four rules:

\[
\text{AssignmentExpression}{}^{[\allowIn \noIn]} \text{ ConditionalExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\allowIn]} = \text{AssignmentExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\allowIn]} \text{ CompoundAssignment AssignmentExpression}{}^{[\allowIn \noIn]}
\]

\[
\text{AssignmentExpression}{}^{[\initial, \allowIn \noIn]} \text{ ConditionalExpression}{}^{[\initial, \allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\initial]} = \text{AssignmentExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\initial]} \text{ CompoundAssignment AssignmentExpression}{}^{[\allowIn \noIn]}
\]

\[
\text{AssignmentExpression}{}^{[\initial, \noIn]} \text{ ConditionalExpression}{}^{[\initial, \noIn]} \\
| \text{LeftSideExpression}{}^{[\initial]} = \text{AssignmentExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\initial]} \text{ CompoundAssignment AssignmentExpression}{}^{[\noIn, \noIn]}
\]

\[
\text{AssignmentExpression}{}^{[\allowIn \noIn]} \text{ ConditionalExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\noIn]} = \text{AssignmentExpression}{}^{[\allowIn \noIn]} \\
| \text{LeftSideExpression}{}^{[\noIn]} \text{ CompoundAssignment AssignmentExpression}{}^{[\noIn, \noIn]}
\]

\[
\text{AssignmentExpression}{}^{[\allowIn \noIn]} \text{ is now an unparametrised nonterminal and processed normally by the grammar.}
\]

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 \([\allowIn \noIn]\).

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:

\[
\text{NonAsteriskOrSlash} \allowIn \text{ UnicodeCharacter} \allowIn \text{ except} \ * \ | \ /
\]

### 5.15 Semantic Actions

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

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

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

\[
\begin{align*}
Digit & \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
Digits & \in Digit \mid Digits Digit \\
Numeral & \in Digits \mid Digits # Digits
\end{align*}
\]

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:

\[
\begin{align*}
\text{Value}[Digit] & : \text{INTEGER} = \text{Digit’s decimal value (an integer between 0 and 9)}. \\
\text{DecimalValue}[Digits] & : \text{INTEGER}; \\
\text{DecimalValue}[Digits] & : \text{INTEGER} = \text{Value}[Digit]; \\
\text{DecimalValue}[Digits_0 # Digits_1 Digit] & = 10 \cdot \text{DecimalValue}[Digits_1] + \text{Value}[Digit]; \\
\text{proc BaseValue}[Digits] (\text{base}: \text{INTEGER}) : \text{INTEGER} \\
\{ \\
\text{[Digits} \mid \text{Digit}) \text{ do} \\
\quad d : \text{INTEGER} \equiv \text{Value}[Digit]; \\
\quad \text{if } d \less base \text{ then return } d \text{ else throw } \text{syntaxError} \text{ end if}; \\
\text{[Digits_0} # Digits_1 Digit) \text{ do} \\
\quad d : \text{INTEGER} \equiv \text{Value}[Digit]; \\
\quad \text{if } d \less base \text{ then return } \text{base} \cdot \text{BaseValue}[Digits_1](\text{base}) + d \text{ else throw } \text{syntaxError} \text{ end if}; \\
\text{end proc}; \\
\text{Value}[Numeral] : \text{INTEGER}; \\
\text{Value}[Numeral] & : \text{INTEGER} = \text{DecimalValue}[Digits]; \\
\text{Value}[Numeral] & : \text{INTEGER} = \text{DecimalValue}[Digits_0 # Digits_1 # Digits_2] \\
\text{begin} \\
\quad \text{base : INTEGER} \equiv \text{DecimalValue}[Digits_2]; \\
\quad \text{if } \text{base} \geq 2 \text{ and } \text{base} \leq 10 \text{ then return } \text{BaseValue}[Digits_1](\text{base}) \text{ else throw } \text{syntaxError} \text{ end if}; \\
\text{end;}
\end{align*}
\]

Action names are written in cursive type. The definition

\[
\text{Value}[Numeral] : \text{INTEGER};
\]

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;
    [Digits1 [] 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:

<table>
<thead>
<tr>
<th>Input</th>
<th>Semantic Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>33#4</td>
<td>15</td>
</tr>
<tr>
<td>30#2</td>
<td>throw syntaxError</td>
</tr>
</tbody>
</table>

5.15.2 Abbreviated Actions

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

Given the sample grammar rule
Expression []
  Subexpression
| Expression * Subexpression
| Subexpression + Subexpression
| this

the notation

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

is an abbreviation for the following:

proc Validate[Expression] (cxt: CONTEXT, env: ENVIRONMENT)
  [Expression [] Subexpression] do Validate[Subexpression](cxt, env);
  [Expression [] Expression * Subexpression] do
    Validate[Expression](cxt, env);
    Validate[Subexpression](cxt, env);
  [Expression [] Subexpression + Subexpression] 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.

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

Id []
  SimpleId
  ComplexId

the notation

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

is an abbreviation for the following:

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

5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

Action[nonterminal]: T;

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

Action[nonterminal [] expansion] = expression;

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

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

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

Action[nonterminal [] expansion]

begin
   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 stepn. 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 stepn.

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 expansionn. 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, ... , paramn: Tn) 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: \[ T = \text{expression}; \]
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: \[ T \square \text{expression}; \]
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 \[ f(\text{param}_1: T_1, \ldots, \text{param}_{m}: T_m): T \]
\[ \text{step}_1; \]
\[ \text{step}_2; \]
\[ \ldots; \]
\[ \text{step}_m; \]
end proc;
This notation defines \( f \) to be a procedure (section 5.13).

6 Source Text

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

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

In string literals, regular expression literals and identifiers, any character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely \( \\text{u} \) 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 \( \text{u}000A \), 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 \( \text{u}000A \) 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 \( \text{n} \) instead of \( \text{u}000A \) to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

6.1 Unicode Format-Control Characters

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

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

7 Lexical Grammar

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

A token is one of the following:
- A keyword token, which is either:
  - One of the reserved words currently used by ECMAScript as, break, case, catch, class, const, continue, default, delete, do, else, export, extends, false, final, finally, for, function, if, import, in, instanceof, is, namespace, new, null, package, private, public, return, static, super, switch, this, throw, true, try, typeof, use, var, void, while, with.
  - One of the reserved words reserved for future use abstract, debugger, enum, goto, implements, interface, native, protected, synchronized, throws, transient, volatile.
- One of the non-reserved words exclude, get, include, set.
- A punctuator token, which is one of !, ==, =, %, &>&, ^=, |=, (), *, +=, *=, +=, +=, =, -=, --, --, . . . .
- A 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:

InputElement = [LineBreak, EndOfInput] ⋐ TOKEN

The lexical grammar has individual characters as its terminal symbols plus the special terminal End, which is appended after the last input character. The lexical grammar defines three goal symbols NextInputElement*, NextInputElement^#, and NextInputElement^#, 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^# if the previous lexed token was a Number or NegatedMinLong, NextInputElement* if the previous token was not a Number or NegatedMinLong and a / should be interpreted as starting a regular expression, and NextInputElement^# 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 \textit{inputElements} be an empty sequence of input elements.
Let \textit{input} be the input sequence of characters. Append a special placeholder \texttt{End} to the end of \textit{input}.
Let \textit{state} be a variable that holds one of the constants \texttt{re}, \texttt{div}, or \texttt{num}. Initialise it to \texttt{re}.

Repeat the following steps until exited:

1. Find the longest possible prefix \( P \) of \textit{input} that is a member of the lexical grammar’s language (see section 5.14).
2. Use the start symbol \texttt{NextInputElement}, \texttt{NextInputElementDiv}, or \texttt{NextInputElementNum} depending on whether \textit{state} is \texttt{re}, \texttt{div}, or \texttt{num}, respectively. If the parse failed, signal a syntax error.
3. Compute the action \texttt{Lex} on the derivation of \( P \) to obtain an input element \( e \).
4. If \( e \) is \texttt{EndOfInput}, then exit the repeat loop.
5. Remove the prefix \( P \) from \textit{input}, leaving only the yet-unprocessed suffix of \textit{input}.
6. Append \( e \) to the end of the \textit{inputElements} sequence.
7. If the \textit{inputElements} sequence does not form a valid sentence prefix of the language defined by the syntactic grammar, then:
   - If \( e \) is not \texttt{LineBreak}, but the next-to-last element of \textit{inputElements} is \texttt{LineBreak}, then insert a \texttt{VirtualSemicolon} terminal between the next-to-last element and \( e \) in \textit{inputElements}.
   - If \textit{inputElements} still does not form a valid sentence prefix of the language defined by the syntactic grammar, signal a syntax error.
8. End if
9. If \( e \) is a \texttt{Number} token, then set \textit{state} to \texttt{num}. Otherwise, if the \textit{inputElements} sequence followed by the terminal / forms a valid sentence prefix of the language defined by the syntactic grammar, then set \textit{state} to \texttt{div}; otherwise, set \textit{state} to \texttt{re}.
10. End repeat
11. If the \textit{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 \textit{inputElements}.

### 7.1 Input Elements

#### Syntax

\begin{align*}
\texttt{NextInputElement} & \quad \texttt{WhiteSpace InputElement} \\
\texttt{NextInputElementDiv} & \quad \texttt{WhiteSpace InputElementDiv} \\
\texttt{NextInputElementNum} & \quad \texttt{lookahead} \{ \texttt{ContinuingIdentifierCharacter, \}} \texttt{WhiteSpace InputElementDiv} \\
\texttt{InputElement} & \quad \texttt{LineBreaks} \\
& \quad | \texttt{IdentifierOrKeyword} \quad \texttt{(IdentifierOrKeyword: 7.5)} \\
& \quad | \texttt{Punctuator} \quad \texttt{(Punctuator: 7.6)} \\
& \quad | \texttt{NumericLiteral} \quad \texttt{(NumericLiteral: 7.7)} \\
& \quad | \texttt{StringLiteral} \quad \texttt{(StringLiteral: 7.8)} \\
& \quad | \texttt{RegExpLiteral} \quad \texttt{(RegExpLiteral: 7.9)} \\
& \quad | \texttt{EndOfInput} \\
\texttt{InputElementDiv} & \quad \texttt{LineBreaks} \\
& \quad | \texttt{IdentifierOrKeyword} \\
& \quad | \texttt{Punctuator} \\
& \quad | \texttt{DivisionPunctuator} \quad \texttt{(DivisionPunctuator: 7.6)} \\
& \quad | \texttt{NumericLiteral} \\
& \quad | \texttt{StringLiteral} \\
& \quad | \texttt{EndOfInput} \\
\texttt{EndOfInput} & \quad \texttt{End} \\
& \quad | \texttt{LineComment} \quad \texttt{(LineComment: 7.4)}
\end{align*}
Semantics

The grammar parameter \$n\$ can be either \$re\$ or \$div\$.

\[
\begin{align*}
\text{Lex}[\text{NextInputElement}^{re}] & : \text{InputElement}; \\
\text{Lex}[\text{NextInputElement}^{div} \text{ WhiteSpace InputElement}^{div}] & = \text{Lex}[\text{InputElement}^{div}]; \\
\text{Lex}[\text{NextInputElement}^{div} \text{ WhiteSpace InputElement}^{div}] & = \text{Lex}[\text{InputElement}^{div}]; \\
\text{Lex}[\text{InputElement}^{num}] & = [\text{lookahead} \{ \text{ContinuingIdentifierCharacter}, \}] \text{ WhiteSpace InputElement}^{div} \text{ WhiteSpace} \\
\text{Lex}[\text{InputElement}^{div} & ]: \text{InputElement}; \\
\text{Lex}[\text{InputElement}^{div} \text{ LineBreaks}] & = \text{LineBreak}; \\
\text{Lex}[\text{InputElement}^{div} \text{ IdentifierOrKeyword}] & = \text{Lex}[\text{IdentifierOrKeyword}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ Punctuator}] & = \text{Lex}[\text{Punctuator}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ DivisionPunctuator}] & = \text{Lex}[\text{DivisionPunctuator}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ NumericLiteral}] & = \text{Lex}[\text{NumericLiteral}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ StringLiteral}] & = \text{Lex}[\text{StringLiteral}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ RegExpLiteral}] & = \text{Lex}[\text{RegExpLiteral}]; \\
\text{Lex}[\text{InputElement}^{div} \text{ EndOfInput}] & = \text{EndOfInput};
\end{align*}
\]

### 7.2 White space

**Syntax**

\[
\begin{align*}
\text{WhiteSpace} & : \\
\text{ «empty»} & | \text{ WhiteSpace WhiteSpaceCharacter} \\
& | \text{ WhiteSpace SingleLineBlockComment} \text{ (SingleLineBlockComment: 7.4)}
\end{align*}
\]

\[
\begin{align*}
\text{WhiteSpaceCharacter} & : \\
\text{ «TAB»} & | \text{ «VT»} & | \text{ «FF»} & | \text{ «SP»} & | \text{ «u00A0»} \\
& | \text{ Any other character in category Zs in the Unicode Character Database}
\end{align*}
\]

**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**

\[
\begin{align*}
\text{LineBreak} & : \\
\text{ LineTerminator} & | \text{ LineComment LineTerminator} \text{ (LineComment: 7.4)} \\
& | \text{ MultiLineBlockComment} \text{ (MultiLineBlockComment: 7.4)}
\end{align*}
\]

\[
\begin{align*}
\text{LineBreaks} & : \\
\text{ LineBreak} & | \text{ LineBreaks WhiteSpace LineBreak} \text{ (WhiteSpace: 7.2)}
\end{align*}
\]

**LineTerminator** : \text{ «LF»} | \text{ «CR»} | \text{ «u2028»} | \text{ «u2029»}

**NOTE** Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section ****).
7.4 Comments

Syntax

LineComment [] / / LineCommentCharacters

LineCommentCharacters []
  «empty»
  | LineCommentCharacters NonTerminator

SingleLineBlockComment [] / * BlockCommentCharacters */

BlockCommentCharacters []
  «empty»
  | BlockCommentCharacters NonTerminatorOrSlash
  | PreSlashCharacters /

PreSlashCharacters []
  «empty»
  | BlockCommentCharacters NonTerminatorOrAsteriskOrSlash
  | PreSlashCharacters /

MultiLineBlockComment [] / * MultiLineBlockCommentCharacters BlockCommentCharacters */

MultiLineBlockCommentCharacters []
  BlockCommentCharacters LineTerminator
  | MultiLineBlockCommentCharacters BlockCommentCharacters LineTerminator (LineTerminator: 7.3)

UnicodeCharacter [] Any Unicode character

NonTerminator [] UnicodeCharacter except LineTerminator

NonTerminatorOrSlash [] NonTerminator except /

NonTerminatorOrAsteriskOrSlash [] NonTerminator except * | /

NOTE  Comments can be either line comments or block comments. Line comments start with a // and continue to the end of the line. Block comments start with /* and end with */. Block comments can span multiple lines but cannot nest.

Except when it is on the last line of input, a line comment is always followed by a LineTerminator. That LineTerminator is not considered to be part of that line comment; it is recognised separately and becomes a LineBreak. A block comment that actually spans more than one line is also considered to be a LineBreak.

7.5 Keywords and Identifiers

Syntax

IdentifierOrKeyword [] IdentifierName
Semantics

\[ \text{Lex}[\text{IdentifierOrKeyword} \ [, \text{IdentifierName}]] : \text{InputElement} \]

\begin{verbatim}
 begin
   id: STRING [] LexName[IdentifierName];
     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;
\end{verbatim}

\textbf{NOTE}  Even though the lexical grammar treats \texttt{exclude}, \texttt{get}, \texttt{include}, and \texttt{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 \texttt{IdentifierName} can never be a keyword if it contains any escape characters, so, for example, one can use \texttt{new} as the name of an identifier by including an escape sequence in it; \texttt{\_new} is one possibility, and \texttt{n\_x65w} is another.

Syntax

\[ \text{IdentifierName} [] \]

\begin{verbatim}
   InitialIdentifierCharacterOrEscape
   | NullEscapes InitialIdentifierCharacterOrEscape
   | IdentifierName ContinuingIdentifierCharacterOrEscape
   | IdentifierName NullEscape

   NullEscapes []
   NullEscape
   | NullEscapes NullEscape

   NullEscape [] \_\_

   InitialIdentifierCharacterOrEscape [] \_ HexEscape
   (HexEscape: 7.8)

   InitialIdentifierCharacter [] UnicodeInitialAlphabetic | $ | _

   UnicodeInitialAlphabetic [] Any character in category \texttt{Lu} (uppercase letter), \texttt{Ll} (lowercase letter), \texttt{Lt} (titlecase letter), \texttt{Lm} (modifier letter), \texttt{Lo} (other letter), or \texttt{Nl} (letter number) in the Unicode Character Database

   ContinuingIdentifierCharacterOrEscape []
   ContinuingIdentifierCharacter
   | \ HexEscape

   ContinuingIdentifierCharacter [] UnicodeAlphanumeric | $ | _

   UnicodeAlphanumeric [] Any character in category \texttt{Lu} (uppercase letter), \texttt{Ll} (lowercase letter), \texttt{Lt} (titlecase letter), \texttt{Lm} (modifier letter), \texttt{Lo} (other letter), \texttt{Nd} (decimal number), \texttt{Nl} (letter number), \texttt{Mn} (non-spacing mark), \texttt{Mc} (combining spacing mark), or \texttt{Pc} (connector punctuation) in the Unicode Character Database
\end{verbatim}
Semantics

LexName[IdentifierName]: STRING;
LexName[IdentifierName] = [LexChar[InitialIdentifierCharacterOrEscape]];
LexName[IdentifierName] = [LexChar[NullEscapes InitialIdentifierCharacterOrEscape]];
LexName[IdentifierName] = LexName[IdentifierName] \ LexChar[InitialIdentifierCharacterOrEscape]

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

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

The characters in the specified categories in version 3.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations; however, conforming ECMAScript implementations may allow additional legal identifier characters based on the category assignment from later versions of Unicode.

NOTE Identifiers are interpreted according to the grammar given in Section 5.16 of version 3.0 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode standard. This standard specifies one departure from the grammar given in the Unicode standard: $ and _ are permitted anywhere in an identifier. $ is intended for use only in mechanically generated code.

Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier. An escape sequence cannot be used to put a character into an identifier that would otherwise be illegal in that position of the identifier.

Two identifiers that are canonically equivalent according to the Unicode standard are not equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.
7.6 Punctuators

Syntax

```
Punctuator []
| !                      | ! =                     | ! = =                   | %                      | % =                     | \&                      | \& \&                    |
| \& \& =                | \& =                    | (                       | )                      | *                       | \* =                    | +                       |
| ++                     | +=                     | ,                       | -                      | -=                      | -=                      | .                       |
| ...                    | :                      | ::                      | ;                      | <                      | \<                      | \< \<                   |
| <=                     | =                      | ==                      | ==                     | >                      | \>                      | \> >                    |
| >=                     | > >                    | > > >                   | > > > =                | ?                      | [                       | ]                       |
| ^=                     | ^ ^                    | ^ ^ =                   | {                      | |                       | |                       |
| | =                    | |                      | |                       | |                       | |                       |
```

```
DivisionPunctuator []
/ [lookahead] { /, + ]
/ =
```

Semantics

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

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

7.7 Numeric literals

Syntax

```
NumericLiteral []
| DecimalLiteral
| HexIntegerLiteral
| DecimalLiteral LetterF
| IntegerLiteral LetterL
| IntegerLiteral LetterU LetterL
```

```
IntegerLiteral []
| DecimalIntegerLiteral
| HexIntegerLiteral
```

```
LetterF [] F | f
```

```
LetterL [] L | l
```

```
LetterU [] U | u
```

```
DecimalLiteral []
| Mantissa
| Mantissa LetterE SignedInteger
```

```
LetterE [] E | e
```

```
Mantissa []
| DecimalIntegerLiteral
| DecimalIntegerLiteral .
| DecimalIntegerLiteral . Fraction
| . Fraction
```
DecimalIntegerLiteral □
  0
  | NonZeroDecimalDigits

NonZeroDecimalDigits □
  NonZeroDigit
  | NonZeroDecimalDigits ASCIIDigit

Fraction □ DecimalDigits

SignedInteger □
  DecimalDigits
  | + DecimalDigits
  | - DecimalDigits

DecimalDigits □
  ASCIIDigit
  | DecimalDigits ASCIIDigit

HexIntegerLiteral □
  0 LetterX HexDigit
  | HexIntegerLiteral HexDigit

LetterX □ \x

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

NonZeroDigit □ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit □ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

Semantics

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

NOTE All digits of hexadecimal literals are significant.

LexNumber[DecimalLiteral]: RATIONAL;
    LexNumber[DecimalLiteral [] Mantissa] = LexNumber[Mantissa];
    LexNumber[DecimalLiteral [] Mantissa LetterE SignedInteger] = LexNumber[Mantissa]10LexNumber[SignedInteger];

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

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

LexNumber[NonZeroDecimalDigits]: INTEGER;
    LexNumber[NonZeroDecimalDigits [] NonZeroDigit] = DecimalValue[NonZeroDigit];
    LexNumber[NonZeroDecimalDigits [] NonZeroDecimalDigits, ASCIIDigit]
        = 10LexNumber[NonZeroDecimalDigits] + DecimalValue[ASCIIDigit];

LexNumber[Fraction [] DecimalDigits]: RATIONAL = LexNumber[DecimalDigits]10NDecimalDigits[DecimalDigits];

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

LexNumber[DecimalDigits]: INTEGER;
    LexNumber[DecimalDigits [] ASCIIDigit] = DecimalValue[ASCIIDigit];
    LexNumber[DecimalDigits [] ASCIIDigit] = 10LexNumber[DecimalDigits] + DecimalValue[ASCIIDigit];

NDecimalDigits[DecimalDigits]: INTEGER;
    NDecimalDigits[DecimalDigits [] ASCIIDigit] = 1;
    NDecimalDigits[DecimalDigits [] ASCIIDigit] = NDecimalDigits[DecimalDigits] + 1;

LexNumber[HexIntegerLiteral]: INTEGER;
    LexNumber[HexIntegerLiteral [] 0 LetterX HexDigit] = HexValue[HexDigit];
    LexNumber[HexIntegerLiteral [] HexIntegerLiteral, HexDigit]
        = 16LexNumber[HexIntegerLiteral] + 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 q can be either single or double.

`StringLiteral [ ]` can be:

- single quotes and string characters:
  - `StringChar[single]`

- double quotes and string characters:
  - `StringChar[double]`

`StringChars[ ]` can be:

- «empty»
- `StringChar[ ] StringChar[single]`
- `StringChar[ ] NullEscape`

`StringEscape[ ]` can be:

- `ControlEscape`
- `ZeroEscape`
- `HexEscape`
- `IdentityEscape`

`IdentityEscape[ ]` can be:

- `_`
- `UnicodeAlphanumeric`

`ControlEscape[ ]` can be:

- `b`
- `f`
- `n`
- `r`
- `t`
- `v`

`ZeroEscape[ ]` can be:

- `0` (lookahead ASCII digit)

`HexEscape[ ]` can be:

- `x` `HexDigit` `HexDigit`
- `u` `HexDigit` `HexDigit` `HexDigit` `HexDigit`

Semantics

`Lex[StringLiteral]: Token;`

- `Lex[StringLiteral]` is a `String` token with the value `LexString[StringChars[single]]`.
- `Lex[StringLiteral]` is a `String` token with the value `LexString[StringChars[double]]`.

`LexString[StringChars[ ]]: String;`

- `Lex[StringChars[ ] «empty»]` is «""».
- `Lex[StringChars[ ] StringChars[single]]` is `LexString[StringChars[ ] StringChar[single]]`.

`LexChar[StringChar[ ]]: Character;`

- `LexChar[StringChar[ ] LiteralStringChar[ ] ]` is `LiteralStringChar[ ]`.
- `LexChar[StringChar[ ] \ StringEscape]` is `LexChar[StringEscape]`.
LexChar[StringEscape]: CHARACTER;
  LexChar[StringEscape [] ControlEscape] = LexChar[ControlEscape];
  LexChar[StringEscape [] ZeroEscape] = LexChar[ZeroEscape];
  LexChar[StringEscape [] HexEscape] = LexChar[HexEscape];
  LexChar[StringEscape [] IdentityEscape] = IdentityEscape;

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

LexChar[ControlEscape]: CHARACTER;
  LexChar[ControlEscape [] b] = ‘BS’;
  LexChar[ControlEscape [] f] = ‘FF’;
  LexChar[ControlEscape [] n] = ‘LF’;
  LexChar[ControlEscape [] r] = ‘CR’;
  LexChar[ControlEscape [] t] = ‘TAB’;
  LexChar[ControlEscape [] v] = ‘VT’;

LexChar[ZeroEscape [] 0 [lookahead {} ASCIIDigit]]: CHARACTER = ‘NUL’;

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

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
  | RegExpFlags NullEscape

(ContinuingIdentifierCharacterOrEscape: 7.5)
(NullEscape: 7.5)

RegExpBody [] [lookahead {} +] RegExpChars /

RegExpChars []
  RegExpChar
  | RegExpChars RegExpChar

(NonTerminator: 7.4)

OrdinaryRegExpChar [] NonTerminator except \ | /
Semantics

Lex[RegExpLiteral] = [RegExpBody] Reflgs]; Token
  = A RegularExpression token with the body LexString[RegExpBody] and flags LexString[RegExpFlags];

LexString[RegExpFlags] = STRING;
LexString[RegExpFlags] = "empty" = ""
LexString[RegExpFlags] = ContinuationIdentifierCharacterOrEscape
  = LexString[RegExpFlags] + LexChar[ContinuationIdentifierCharacterOrEscape]];

LexString[RegExpChars] = [lookahead] [RegExpChars] = LexString[RegExpChars];
LexString[RegExpChars] = LexString[RegExpChars] + LexString[RegExpChars];
LexString[RegExpChars] = LexString[RegExpChars] + LexString[RegExpChars];
LexString[RegExpChars] = LexString[RegExpChars] + LexString[RegExpChars];

LexString[RegExpChar] = OrdinaryRegExpChar = [OrdinaryRegExpChar];
LexString[RegExpChar] = \ NonTerminator = [\ NonTerminator];

(Note that the result string has two characters)

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

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

8 Program Structure

8.1 Packages

8.2 Scopes

9 Data Model

This chapter describes the essential state held in various ECMAScript objects. This state is presented abstractly using the formalisms from chapter 5. Much of the state held in these objects is observable by ECMAScript programmers only indirectly, and implementations are encouraged to implement these objects in more efficient ways as long as the observable behaviour is the same as described here.

9.1 Objects

An object is a first-class data value visible to ECMAScript programmers. Every object is either undefined, null, a Boolean, a signed or unsigned 64-bit integer, a single or double-precision floating-point number, a character, a string, a namespace, a compound attribute, a class, a simple instance, a method closure, a date, a regular expression, or a package object. These kinds of objects are described in the subsections below.
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.

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

```
PRIMITIVEOBJECT = Undefined [] Null [] Boolean [] Long [] ULong [] FLOAT32 [] FLOAT64 [] Character [] CHAR16 [] STRING [] NAMESPACE [] COMPOUNDATTRIBUTE [] CLASS [] SIMPLEINSTANCE [] METHODCLOSURE [] DATE [] REGEXP [] PACKAGE;
```

A `BINDINGOBJECT` is an object that can bind local properties:

```
BINDINGOBJECT = Class [] SIMPLEINSTANCE [] REGEXP [] DATE [] PACKAGE;
```

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

```
OBJECTOPT = OBJECT [] {none};
```

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

```
INTEGEROPT = INTEGER [] {none};
```

9.1.1 Undefined

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

```
UNDEFINED = {undefined}
```

9.1.2 Null

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

```
NULL = {null}
```

9.1.3 Booleans

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

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

```
BOOLEANOPT = BOOLEAN [] {none};
```

9.1.4 Numbers

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

9.1.5 Strings

The semantic domain `STRING` consists of all representable strings. See section 5.9.

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

```
STRINGOPT = STRING [] {none}
```

9.1.6 Namespaces

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td><code>NAMESPACE</code></td>
<td>The namespace qualifier</td>
</tr>
<tr>
<td>id</td>
<td><code>STRING</code></td>
<td>The name</td>
</tr>
</tbody>
</table>

The semantic notation `ns::id` is a shorthand for `QUALIFIEDNAME[namespace: ns, id: id]`.

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

`MULTINAME = QUALIFIEDNAME[]`

### 9.1.7 Compound attributes

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespaces</td>
<td><code>NAMESPACE[]</code></td>
<td>The set of namespaces contained in this attribute</td>
</tr>
<tr>
<td>explicit</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if the <code>explicit</code> attribute has been given</td>
</tr>
<tr>
<td>enumerable</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if the <code>enumerable</code> attribute has been given</td>
</tr>
<tr>
<td>dynamic</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if the <code>dynamic</code> attribute has been given</td>
</tr>
<tr>
<td>categorymemberMod</td>
<td><code>PROPERTYCATEGORYMEMBERMODIFIER</code></td>
<td><code>static</code>, <code>virtual</code>, or <code>final</code> if one of these attributes has been given; <code>none</code> if not.</td>
</tr>
<tr>
<td>overrideMod</td>
<td><code>OVERRIDEDECORATOR</code></td>
<td><code>true, false, or undefined</code> if the <code>override</code> attribute with one of these arguments was given; <code>true</code> if the attribute <code>override</code> without arguments was given; <code>none</code> if the <code>override</code> attribute was not given.</td>
</tr>
<tr>
<td>prototype</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if the <code>prototype</code> attribute has been given</td>
</tr>
<tr>
<td>unused</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if the <code>unused</code> attribute has been given</td>
</tr>
</tbody>
</table>

**NOTE** An implementation that supports host-defined attributes will add other fields to the tuple above.

`ATTRIBUTE` consists of all attributes and attribute combinations, including Booleans and single namespaces:

```
ATTRIBUTE = BOOLEAN □ NAMESPACE □ COMPOUNDATTRIBUTE
```

`ATTRIBUTEOptNotFalse` consists of `none` as well as all attributes and attribute combinations except for `false`:

```
ATTRIBUTEOptNotFalse = {none, true} □ NAMESPACE □ COMPOUNDATTRIBUTE
```

### 9.1.8 Classes

Programmer-visible class objects are represented as `CLASS` records (see section 5.11) with the fields below.
<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Map of qualified names to members—singleton properties defined in this class (see section *****)</td>
</tr>
<tr>
<td>instanceProperties</td>
<td>INSTANCEPROPERTY{}</td>
<td>Map of qualified names to instance—properties defined or overridden in this class</td>
</tr>
<tr>
<td>super</td>
<td>CLASSOPT</td>
<td>This class’s immediate superclass or null if none</td>
</tr>
<tr>
<td>prototype</td>
<td>OBJECTOPT</td>
<td>The default archetype of new instances of this class</td>
</tr>
<tr>
<td>complete</td>
<td>BOOLEAN</td>
<td>true after all members of this class have been added to this CLASS record</td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>This class’s name</td>
</tr>
<tr>
<td>typeofString</td>
<td>STRING</td>
<td>A string to return if typeof is invoked on this class’s instances</td>
</tr>
<tr>
<td>privateNamespace</td>
<td>NAMESPACE</td>
<td>This class’s private namespace</td>
</tr>
<tr>
<td>dynamic</td>
<td>BOOLEAN</td>
<td>true if this class or any of its ancestors was defined with the dynamic attribute</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this class cannot be subclassed</td>
</tr>
<tr>
<td>defaultValue</td>
<td>OBJECTOPT</td>
<td>When a variable whose type is this class is defined but not explicitly initialized, the variable’s initial value defaultvalue, which must be instance of this class. The class Never has no values, so that class’s (and others) defaultvalue is none.</td>
</tr>
<tr>
<td>defaultHint</td>
<td>HINT</td>
<td>The default hint to use when converting an instance of this class to a primitive</td>
</tr>
<tr>
<td>hasProperty</td>
<td>OBJECT[] CLASS[] OBJECT[] BOOLEAN [] PHASE[] BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>bracketRead</td>
<td>OBJECT[] CLASS[] OBJECT[] BOOLEAN [] PHASE[] OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>bracketWrite</td>
<td>OBJECT[] CLASS[] OBJECT[] OBJECT[] Boolean [] {run} [] {none, ok}</td>
<td></td>
</tr>
<tr>
<td>bracketDelete</td>
<td>OBJECT[] CLASS[] OBJECT[] {run} [] BOOLEANOPT</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>OBJECT[] CLASS[] MULTINAME [] ENVIRONMENTOPT[] BOOLEAN [] PHASE[] OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>OBJECT[] CLASS[] MULTINAME [] ENVIRONMENTOPT[] BOOLEAN [] OBJECT[] BOOLEAN [] {run} [] {none, ok}</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>OBJECT[] CLASS[] MULTINAME [] ENVIRONMENTOPT[] {run}</td>
<td></td>
</tr>
</tbody>
</table>
### 9.1.9 Simple Instances

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Map of qualified names to local properties (including dynamic properties, if any) of this instance</td>
</tr>
<tr>
<td>archetypeSuper</td>
<td>OBJECTOPT</td>
<td>Optional link to the next object in this instance’s prototype archetype chain</td>
</tr>
</tbody>
</table>
sealed  BOOLEAN

type  CLASS

slots  SLOT{}

call  (OBJECT [] SIMPLEINSTANCE
      [] OBJECT [] PHASE
      [] OBJECT) [] {none}

construct  (SIMPLEINSTANCE [] OBJECT[
      [PHASE [] OBJECT]
      [none]}

env  ENVIRONMENTOPT

9.1.1 Slots

A SLOT record (see section 5.11) has the fields below and describes the value of one fixed property of one instance.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>INSTANCEVARIABLE</td>
<td>The instance variable whose value this slot carries</td>
</tr>
<tr>
<td>value</td>
<td>OBJECTOPT OBJECTU</td>
<td>This fixed property’s current value; uninitialised none if</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the fixed property is an uninitialised constant</td>
</tr>
</tbody>
</table>

9.1.10 Uninstantiated Functions

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>CLASS</td>
<td>Values to be transferred into the generated SIMPLEINSTANCE’s corresponding fields</td>
</tr>
<tr>
<td>length</td>
<td>INTEGER</td>
<td>The value to store in the generated SIMPLEINSTANCE’s length property</td>
</tr>
</tbody>
</table>
| call  | {OBJECT [] SIMPLEINSTANCE
     [] OBJECT [] PHASE
     [] OBJECT) [] {none} | Values to be transferred into the generated SIMPLEINSTANCE’s corresponding fields |
| construct  | {SIMPLEINSTANCE [] OBJECT[
     [PHASE [] OBJECT]
     [none] | |
| instantiations | SIMPLEINSTANCE{} | Set of prior instantiations. (This set serves only to precisely specify the closure sharing optimization and would not be needed in any actual implementation.) |

9.1.11 Method Closures

A METHODCLOSURE tuple (see section 5.10) has the fields below and describes an instance method with a bound this value.
### 9.1.12 Dates

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>archetype super</td>
<td>OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>timeValue</td>
<td>INTEGER</td>
<td>The date expressed as a count of milliseconds from January 1, 1970 UTC</td>
</tr>
</tbody>
</table>

### 9.1.13 Regular Expressions

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>archetype super</td>
<td>OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>STRING</td>
<td>This regular expression’s source pattern</td>
</tr>
<tr>
<td>lastIndex</td>
<td>INTEGER</td>
<td>The string position at which to start the next regular expression match</td>
</tr>
<tr>
<td>global</td>
<td>BOOLEAN</td>
<td><code>true</code> if the regular expression flags included the flag <code>g</code></td>
</tr>
<tr>
<td>ignoreCase</td>
<td>BOOLEAN</td>
<td><code>true</code> if the regular expression flags included the flag <code>i</code></td>
</tr>
<tr>
<td>multiline</td>
<td>BOOLEAN</td>
<td><code>true</code> if the regular expression flags included the flag <code>m</code></td>
</tr>
</tbody>
</table>

### 9.1.14 Packages and Global Objects

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>archetype super</td>
<td>OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>This package’s name</td>
</tr>
<tr>
<td>initialize</td>
<td>() ⟦ () ⟧ {none, busy}</td>
<td>A procedure to initialize this package</td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>internalNamespace</td>
<td>NAMESPACE</td>
<td>This package’s or global object’s internal namespace</td>
</tr>
</tbody>
</table>

### 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.
Some

is a place where a value may be read or written. A

The value of expr to which the super subexpression was applied; if expr wasn’t given, defaults to the value of this. The value of instance is always an instance of one of the limit class’s descendants.

The immediate superclass of the class inside which the super subexpression was applied.

A

is the result of a subexpression that can produce either an Object or a LimitedInstance:

ObjOptionalLimit = Object [] LimitedInstance

9.3 References

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

Reference = LexicalReference [] DotReference [] BracketReference;

Some subexpressions evaluate to an ObjOrRef, which is either an Object (also known as an rvalue) or a Reference. Attempting to use an ObjOrRef that is an rvalue as the destination of an assignment produces an error.

ObjOrRef = Object [] Reference

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

Field | Contents | Note
--- | --- | ---
env | Environment | The environment in which the reference was created.
variableMultiname | Multiname | A nonempty set of qualified names to which this reference can refer
strict | Boolean | true if strict mode was in effect at the point where the reference was created

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

Field | Contents | Note
--- | --- | ---
base | Object | The object whose property was referenced (a in the examples above).
limit | Class | The most specific class to consider when searching for properties of the object a. Normally limit is a’s class, but can be one of that class’s ancestors if a is a super expression.

multinamepropertyMultiname | Multiname | A nonempty set of qualified names to which this reference can refer (b qualified with the namespace q or all currently open namespaces in the example above)

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

Field | Contents | Note
--- | --- | ---
base | Object | The object whose property was referenced (a in the examples above).
limit | Class | The most specific class to consider when searching for properties of the object a. Normally limit is a’s class, but can be one of that class’s ancestors if a is a super expression.
class, but can be one of that class’s ancestors if \( a \) is a **super** expression.

\[
\text{args} \quad \text{OBJECT[]} \quad \text{The list of arguments between the brackets (x or \( x, y \) in the examples above)}
\]

### 9.4 Phases of evaluation

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

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

\[
\text{Phase} = \{ \text{compile}, \text{run} \}
\]

### 9.5 Contexts

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if strict mode (see *****) is in effect</td>
</tr>
<tr>
<td>openNamespaces</td>
<td>NAMESPACE{}</td>
<td>The set of namespaces that are open at this point. The <strong>public</strong> namespace is always a member of this set.</td>
</tr>
</tbody>
</table>

### 9.6 Labels

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

\[
\text{Label} = \text{STRING} \begin{cases}
\text{default}
\end{cases}
\]

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>breakTargets</td>
<td>LABEL{}</td>
<td>The set of labels that are valid destinations for a <strong>break</strong> statement</td>
</tr>
<tr>
<td>continueTargets</td>
<td>LABEL{}</td>
<td>The set of labels that are valid destinations for a <strong>continue</strong> statement</td>
</tr>
</tbody>
</table>

### 9.7 Semantic Exceptions

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

\[
\text{SemanticException} = \text{OBJECT} \begin{cases}
\text{CONTROLTRANSFER}
\end{cases};
\text{CONTROLTRANSFER} = \text{BREAK} \begin{cases}
\text{CONTINUE} \begin{cases}
\text{RETURN}
\end{cases}
\end{cases};
\]

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

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT</td>
<td>The value produced by the last statement to be executed before the <strong>break</strong></td>
</tr>
<tr>
<td>label</td>
<td>LABEL</td>
<td>The label that is the target of the <strong>break</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT</td>
<td>The value of the expression in the <strong>return</strong> statement or <strong>undefined</strong> if omitted</td>
</tr>
</tbody>
</table>

### 9.8 Function Support

The **FUNCTION_KIND** semantic domain encodes a general kind of a function:

```
FUNCTION_KIND = {plainFunction, uncheckedFunction, prototypeFunction, instanceFunction, constructorFunction};
```

These kinds represent the following:

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

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

```
STATIC_FUNCTION_KIND = {plainFunction, uncheckedFunction, prototypeFunction};
```

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

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

### 9.9 Environment Frames

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 **SYSTEM_FRAME**. The next to last frame is always a **PACKAGE**. A **WITH_FRAME** is always preceded by a **LOCAL_FRAME**, so the first frame is never a **WITH_FRAME**.

```
ENVIRONMENT = FRAME[]
```

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

```
ENVIRONMENT_OPT = ENVIRONMENT [] {none};
```

A frame contains bindings defined at a particular scope in a program. A frame is either the **top-level system frame**, a package, a function parameter frame, a class, a local (block) frame, or a **with** statement frame:
Some frames hold the runtime values of variables and other definitions; these frames are called *instantiated frames*. Other frames, called *uninstantiated frames*, are used as templates for making (instantiating) instantiated frames. The static analysis done by `Validate` generates instantiated frames for a few top-level scopes and uninstantiated frames for other scopes; the `preinst` parameter to `Validate` governs whether it generates instantiated or uninstantiated frames.

### 9.9.1 System Frame

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>localBindings</code></td>
<td><code>LOCALBINDING{}</code></td>
<td>Map of qualified names to definitions in this frame</td>
</tr>
</tbody>
</table>

### 9.9.2 Function Parameter Frames

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>localBindings</code></td>
<td><code>LOCALBINDING{}</code></td>
<td>Map of qualified names to definitions in this function</td>
</tr>
<tr>
<td><code>kind</code></td>
<td><code>FUNCTION_KIND</code></td>
<td>See section 9.8</td>
</tr>
<tr>
<td><code>handling</code></td>
<td><code>HANDLING</code></td>
<td>See section 9.8</td>
</tr>
<tr>
<td><code>callsSuperconstructor</code></td>
<td><code>BOOLEAN</code></td>
<td>A flag that indicates whether a call to the superclass’s constructor has been detected during static analysis of a class constructor. Always <code>false</code> if <code>kind</code> is not <code>constructorFunction</code>.</td>
</tr>
<tr>
<td><code>superconstructorCalled</code></td>
<td><code>BOOLEAN</code></td>
<td>If <code>kind</code> is a <code>constructorFunction</code>, this flag indicates whether the superclass’s constructor has been called yet during execution of this constructor. Always <code>true</code> if <code>kind</code> is not <code>constructorFunction</code>.</td>
</tr>
<tr>
<td><code>this</code></td>
<td><code>OBJECTOPT</code></td>
<td>The value of <code>this</code>; <code>none</code> if this function doesn’t define <code>this</code> or it defines <code>this</code> but the value is not available because this function hasn’t been called yet</td>
</tr>
<tr>
<td><code>parameters</code></td>
<td><code>PARAMETER[]</code></td>
<td>List of this function’s parameters</td>
</tr>
<tr>
<td><code>rest</code></td>
<td><code>VARIABLEOPT</code></td>
<td>The parameter variable for collecting any extra arguments that may be passed or <code>none</code> if no extra arguments are allowed</td>
</tr>
<tr>
<td><code>returnType</code></td>
<td><code>CLASS</code></td>
<td>The function’s declared return type, which defaults to <code>Object</code> if not provided</td>
</tr>
</tbody>
</table>

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

```
PARAMETERFRAMEOPT = PARAMETERFRAME [] {none};
```

### 9.9.2.19.9.1 Parameters

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var</code></td>
<td><code>VARIABLE [] DYNAMICVAR</code></td>
<td>The local variable that will hold this parameter’s value</td>
</tr>
<tr>
<td><code>default</code></td>
<td><code>OBJECTOPT</code></td>
<td>This parameter’s default value; if <code>none</code>, this parameter is required</td>
</tr>
</tbody>
</table>
9.9.39.9.2 Local Frames

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBinding</td>
<td>LOCALBINDING{}</td>
<td>Map of qualified names to definitions in this frame</td>
</tr>
</tbody>
</table>

9.9.49.9.3 With Frames

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECTOPT</td>
<td>The value of the with statement’s expression or none if not evaluated yet</td>
</tr>
</tbody>
</table>

9.10 Environment Bindings

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

\[
\text{ACCESS} = \{\text{read}, \text{write}\};
\]

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

\[
\text{ACCESSSET} = \{\text{read}, \text{write}, \text{readWrite}\};
\]

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

\[
\text{PROPERTYOPT} = \text{SingletonProperty} \[] \text{InstanceProperty} \[] \{\text{none}\};
\]

9.10.1 Static Bindings

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname</td>
<td>QUALIFIEDNAME</td>
<td>The qualified name bound by this binding</td>
</tr>
<tr>
<td>accesses</td>
<td>ACCESSSET</td>
<td>Accesses for which this member property is visible</td>
</tr>
<tr>
<td>explicit</td>
<td>BOOLEAN</td>
<td>true if this binding should not be imported into the global scope</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this binding should be visible in a for-in statement</td>
</tr>
<tr>
<td>content</td>
<td>SingletonPROPERTYLOCALMEMBER</td>
<td>The member property to which this qualified name was bound</td>
</tr>
</tbody>
</table>

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

\[
\text{SingletonPROPERTYLOCALMEMBER} = \{\text{forbidden}\} \[\text{VARIABLE}\] \[\text{DYNAMICVAR}\] \[\text{GETTER}\] \[\text{SETTER}\];
\]

\[
\text{SingletonPROPERTYOPTLOCALMEMBEROPT} = \text{SingletonPROPERTYLOCALMEMBER} \[\text{none}\];
\]
A **forbidden** static member **singleton property** is one that must not be accessed because there exists a definition for the same qualified name in a more local block.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>CLASS</td>
<td>Type of values that may be stored in this variable</td>
</tr>
<tr>
<td>value</td>
<td>VARIABLEVALUE</td>
<td>This variable’s current value; <strong>future</strong> if the variable has not been declared yet; <strong>uninitialised</strong> if the variable must be written before it can be read</td>
</tr>
<tr>
<td>immutable</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this variable’s value may not be changed once set</td>
</tr>
<tr>
<td>setup</td>
<td>() □ CLASSOPT</td>
<td>A semantic procedure that performs the <strong>Setup</strong> action on the variable or constant definition. <strong>none</strong> if the action has already been performed; <strong>busy</strong> if the action is in the process of being performed and should not be reentered.</td>
</tr>
<tr>
<td></td>
<td>□ {none, busy}</td>
<td>A semantic procedure that computes a variable’s initialiser specified by the programmer. <strong>none</strong> if no initialiser was given or if it has already been evaluated; <strong>busy</strong> if the initialiser is being evaluated now and should not be reentered.</td>
</tr>
<tr>
<td>initializer</td>
<td>INITIALIZER</td>
<td>The environment to provide to <strong>initializer</strong> if this variable is a compile-time constant</td>
</tr>
<tr>
<td>initializerEnv</td>
<td>ENVIRONMENT</td>
<td>The environment bound to this getter; <strong>none</strong> if not yet instantiated</td>
</tr>
</tbody>
</table>

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

\[
\text{VARIABLEOPT} = \text{VARIABLE} \cup \{\text{none}\};
\]

A variable’s value can be either an object, **none** (used when the variable has not been initialised yet), or an uninstantiated function (compile time only):

\[
\text{VARIABLEVALUE} = \{\text{none}\} \cup \text{OBJECT} \cup \text{UNINSTANTIATEDFUNCTION};
\]

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

\[
\text{INITIALIZER} = \text{ENVIRONMENT} \cup \text{PHASE} \cup \text{OBJECT};
\]

\[
\text{INITIALIZERP} = \text{INITIALIZER} \cup \{\text{none}\};
\]

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT □ UninstantiatedFunction</td>
<td>This variable’s current value; may be an uninstantiated function at compile time</td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this variable cannot be deleted using the <strong>delete</strong> operator</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>ENVIRONMENT □ PHASE □ OBJECT</td>
<td>A procedure to call to read the value, passing it the environment from the <strong>env</strong> field below and the current mode of expression evaluation</td>
</tr>
<tr>
<td>env</td>
<td>ENVIRONMENTOPT</td>
<td>The environment bound to this getter; <strong>none</strong> if not yet instantiated</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>OBJECT □ ENVIRONMENT □ PHASE □ ()</td>
<td>A procedure to call to write the value, passing it the new value, the environment from the <strong>env</strong> field below, and the current mode of expression evaluation</td>
</tr>
</tbody>
</table>
env  ENVIRONMENTOpt  The environment bound to this setter; none if not yet instantiated

9.10.2 Instance Bindings

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

\[
\text{INSTANCEPROPERTY} = \text{INSTANCEPROPERTY} \oplus \text{INSTANCEPROPERTY} \oplus \text{INSTANCEPROPERTY} \oplus \text{INSTANCEPROPERTY}.
\]

An INSTANCEVariable record (see section 5.11) has the fields below and describes one instance variable or constant definition. This record is also used as a key to look up an instance’s SLOT (see section 9.1.9.1).

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this instance variable</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this instance variable may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this instance variable’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>type</td>
<td>CLASS</td>
<td>Type of values that may be stored in this variable</td>
</tr>
<tr>
<td>defaultValue</td>
<td>OBJECTOpt</td>
<td>This variable’s default value; none if not provided</td>
</tr>
<tr>
<td>immutable</td>
<td>BOOLEAN</td>
<td>true if this variable’s value may not be changed once set</td>
</tr>
</tbody>
</table>

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

\[
\text{INSTANCEVariableOpt} = \text{INSTANCEVariable} \oplus \text{none}.
\]

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this instance method</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this instance method may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this instance method’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>signature</td>
<td>PARAMETERFRAME</td>
<td>This method’s signature encoded in the PARAMETERFRAME’s parameters, rest, and returnType fields</td>
</tr>
<tr>
<td>length</td>
<td>INTEGER</td>
<td>The instance method’s preferred number of arguments</td>
</tr>
</tbody>
</table>
| call       | OBJECT \[OBJECT[] \[PHASE \[OBJECT \]

A procedure to call when this instance method is invoked. The procedure takes a this OBJECT, a list of argument OBJECTs, and a PHASE (see section 9.4) and produces an OBJECT result.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this getter</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this getter may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this getter’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>signature</td>
<td>PARAMETERFRAME</td>
<td>This getter’s signature encoded in the PARAMETERFRAME’s parameters, rest, and returnType fields</td>
</tr>
<tr>
<td>call</td>
<td>OBJECT [PHASE [OBJECT ] A procedure to call to read the value, passing it the this value and the current mode of expression evaluation</td>
<td></td>
</tr>
</tbody>
</table>
An **INSTANCESETTER** record (see section 5.11) has the fields below and describes one instance setter definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this setter</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this setter may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this setter’s <code>public</code> name should be visible in a <code>for-in</code> statement</td>
</tr>
<tr>
<td>signature</td>
<td>PARAMETERFRAME</td>
<td>This setter’s signature encoded in the <code>PARAMETERFRAME</code>’s <code>parameters</code>, <code>rest</code>, and <code>returnType</code> fields</td>
</tr>
<tr>
<td>call</td>
<td>OBJECT □ OBJECT □ PHASE □ ()</td>
<td>A procedure to call to write the value, passing it the <code>this</code> value, the value being written, and the current mode of expression evaluation</td>
</tr>
</tbody>
</table>

### 9.11 Miscellaneous

#### 9.11.1 Extended Integers and Rationals

An extended integer is an integer or one of the tags `+∞`, `−∞`, or `NaN`:

\[
\text{EXTENDEDINTEGER} = \text{INTEGER} □ \{+∞, −∞, NaN\}.
\]

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

\[
\text{EXTENDEDRATIONAL} = (\text{RATIONAL} − \{0\}) □ \{+zero, −zero, +∞, −∞, NaN\}.
\]

#### 9.11.2 Order

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

\[
\text{ORDER} = \{\text{less}, \text{equal}, \text{greater}, \text{unordered}\};
\]

#### 9.11.3 Hints

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

\[
\text{HINT} = \{\text{hintString}, \text{hintNumber}\};
\]

\[
\text{HINTOPT} = \text{HINT} □ \{\text{none}\};
\]

### 10 Data Operations

This chapter describes core algorithms defined on the values in chapter 9. The algorithms here are not ECMAScript language constructs 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.

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

`signedWrap32(i)` returns `i` converted to a value between `−2^{31}` and `2^{31} − 1` inclusive, wrapping around modulo `2^{32}` if necessary.
proc signedWrap32(i: INTEGER): {–2^{31} ... 2^{31} – 1}
  j: INTEGER [] bitwiseAnd(i, 0xFFFFFFFF);
  if j ≥ 2^{31} then j [] j – 2^{32} end if;
  return j
end proc;

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

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

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

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

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

proc truncateToInteger(x: GENERALNUMBER): INTEGER
  case x of
    {NaN_{32}, NaN_{64}, +\infty_{32}, +\infty_{64}, -\infty_{32}, -\infty_{64}} do return 0;
    FINITEFLOAT32 do return truncateFiniteFloat32(x);
    FINITEFLOAT64 do return truncateFiniteFloat64(x);
    LONG [] ULONG do return x.value
  end case
end proc;

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

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

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

  case x of
  | NaN, NaN, +\infty, -\infty, 0, +0, -0 | do return none;
  | +zero, -zero | do return 0;

  end case

  proc integerToULong(i: INTEGER): long
    if -2^{63} \leq i < 2^{64} then return \text{i}_{long}
    else return \text{i}_{long} + \text{i}_{64}
    end if

  end proc

  proc integerToLong(i: INTEGER): long
    if 0 \leq i < 2^{64} then return \text{i}_{long}
    else return \text{i}_{long} + \text{i}_{64}
    end if

  end proc

  proc rationalToULong(q: RATIONAL): long
    if q \notin INTEGER then return integerToULong(q)
    else
      if \abs{q} \leq 2^{33} then return \text{q}_{64}
      else
        if \text{q} < 2^{-63} \text{ or } \text{q} \geq 2^{64} \text{ then return } \text{q}_{64}
        else
          Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
          note -2^{63} \leq i \leq 2^{64} - 1;
          if i < 2^{63} then return \text{i}_{long} else return \text{i}_{long} end if
        end if
      end if
    end if

  end proc

proc rationalToLong(q: RATIONAL): long
  if q \notin INTEGER then return integerToLong(q)
  else
    if \abs{q} \leq 2^{33} then return \text{q}_{64}
    else
      if \text{q} < 2^{-63} \text{ or } \text{q} \geq 2^{64} \text{ then return } \text{q}_{64}
      else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        note -2^{63} \leq i \leq 2^{64} - 1;
        if i \geq 0 then return \text{i}_{long} else return \text{i}_{long} end if
      end if
    end if
  end if

end proc

integerToLong(i) converts i to the first of the types \text{LONG}, \text{ULONG}, or \text{FLOAT64} that can contain the value i. If necessary, the \text{FLOAT64} result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

proc integerToULong(i: INTEGER): long
  if -2^{63} \leq i \leq 2^{63} - 1 then return \text{i}_{long}
  elseif 2^{63} \leq i \leq 2^{64} - 1 then return \text{i}_{long}
  else return \text{i}_{long} + \text{i}_{64}
  end if

end proc

integerToLong(i) converts i to the first of the types \text{ULONG}, \text{LONG}, or \text{FLOAT64} that can contain the value i. If necessary, the \text{FLOAT64} result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

proc rationalToULong(q: RATIONAL): long
  if q \notin INTEGER then return integerToULong(q)
  else
    if \abs{q} \leq 2^{33} then return \text{q}_{64}
    else
      if \text{q} < 2^{-63} \text{ or } \text{q} \geq 2^{64} \text{ then return } \text{q}_{64}
      else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        note -2^{63} \leq i \leq 2^{64} - 1;
        if i \geq 0 then return \text{i}_{long} else return \text{i}_{long} end if
      end if
    end if
  end if

end proc

rationalToLong(q) converts \text{q} to one of the types \text{LONG}, \text{ULONG}, or \text{FLOAT64}, whichever one can come the closest to representing the true value of \text{q}. If several of these types can come equally close to the value of \text{q}, then one of them is chosen according to the algorithm below.

proc rationalToLong(q: RATIONAL): long
  if q \notin INTEGER then return integerToLong(q)
  else
    if \abs{q} \leq 2^{33} then return \text{q}_{64}
    else
      if \text{q} < 2^{-63} \text{ or } \text{q} \geq 2^{64} \text{ then return } \text{q}_{64}
      else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        note -2^{63} \leq i \leq 2^{64} - 1;
        if i \geq 0 then return \text{i}_{long} else return \text{i}_{long} end if
      end if
    end if
  end if

end proc

rationalToLong(q) converts \text{q} to one of the types \text{ULONG}, \text{LONG}, or \text{FLOAT64}, whichever one can come the closest to representing the true value of \text{q}. If several of these types can come equally close to the value of \text{q}, then one of them is chosen according to the algorithm below.

proc rationalToLong(q: RATIONAL): long
  if q \notin INTEGER then return integerToLong(q)
  else
    if \abs{q} \leq 2^{33} then return \text{q}_{64}
    else
      if \text{q} < 2^{-63} \text{ or } \text{q} \geq 2^{64} \text{ then return } \text{q}_{64}
      else
        Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
        note -2^{63} \leq i \leq 2^{64} - 1;
        if i \geq 0 then return \text{i}_{long} else return \text{i}_{long} end if
      end if
    end if
  end if

end proc

rationalToLong(q) converts \text{q} to one of the types \text{ULONG}, \text{LONG}, or \text{FLOAT64}, whichever one can come the closest to representing the true value of \text{q}. If several of these types can come equally close to the value of \text{q}, then one of them is chosen according to the algorithm below.
proc extendedRationalToFloa$632(q: EXTENDED$632): FLOAT64$632$32
  case q of
    RATIONAL do return q$632$32$632;  
    {+zero} do return +zero$632$32$632;  
    {−zero} do return −zero$632$32$632;  
    {+∞} do return +∞$632$32$632;  
    {−∞} do return −∞$632$32$632;  
    {NaN} do return NaN$632$32$632;
  end case
end proc;

proc extendedRationalToFloa$64(q: EXTENDED$632): FLOAT64$64$632$64
  case q of
    RATIONAL do return q$64$632$64$64;  
    {+zero} do return +zero$64$632$64$64;  
    {−zero} do return −zero$64$632$64$64;  
    {+∞} do return +∞$64$632$64$64;  
    {−∞} do return −∞$64$632$64$64;  
    {NaN} do return NaN$64$632$64$64;
  end case
end proc;

toRational(x) returns the exact RATIONAL value of x.
proc toRational(x: FINITE$632): RATIONAL$632$632$632
  case x of
    {+zero} +zero$632$632$632, −zero$632$632$632, −zero$632$632$632 do return 0;  
    NONZERO$632$632$632 □ NONZERO$64$632$64$64 □ LONG □ ULONG do return x.value
  end case
end proc;

toFloat32(x) converts x to a FLOAT32, using the IEEE 754 “round to nearest” mode.
proc toFloat32(x: GENERAL$632): FLOAT32$632$632$632
  case x of
    LONG □ ULONG do return (x.value)$632$632;  
    FLOAT32 do return x;  
    {−∞}$64$632$64$64 do return −∞$632$632$632;  
    {−zero}$64$632$64$64 do return −zero$632$632$632;  
    {+zero}$64$632$64$64 do return +zero$632$632$632;  
    {+∞}$64$632$64$64 do return +∞$632$632$632;  
    {NaN}$64$632$64$64 do return NaN$632$632$632;  
    NONZERO$64$632$64$64 do return (x.value)$632$632;
  end case
end proc;

toFloat64(x) converts x to a FLOAT64, using the IEEE 754 “round to nearest” mode.
proc toFloat64(x: GENERAL$64): FLOAT64$64$64$64
  case x of
    LONG □ ULONG do return (x.value)$64$64$64;  
    FLOAT32 do return float32ToFloa$64(x);  
    FLOAT64 do return x
  end case
end proc;

generalNumberCompare(x, y) compares x with y using the IEEE 754 rules and returns less if x<y, equal if x=y, greater if x>y, or unordered if either x or y is a NaN. The comparison is done using the exact values of x and y, even if they have different types. Positive infinities compare equal to each other and greater than any other non-NaN values. Negative infinities compare equal to each other and less than any other non-NaN values. Positive and negative zeroes compare equal to each other.
proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
  if x ∈ {NaN32, NaN64} or y ∈ {NaN32, NaN64} then return unordered
  elsif x ∈ {+∞32, +∞64} and y ∈ {+∞32, +∞64} then return equal
  elsif x ∈ {–∞32, –∞64} and y ∈ {–∞32, –∞64} then return equal
  elsif x ∈ {+∞32, +∞64} or y ∈ {–∞32, –∞64} then return greater
  elsif x ∈ {–∞32, –∞64} or y ∈ {+∞32, +∞64} then return less
  else
    xr: RATIONAL = toRational(x);
    yr: RATIONAL = toRational(y);
    if xr < yr then return less
    elsif xr > yr then return greater
    else return equal
  end if
end proc;

10.2 Character Utilities

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

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

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

proc stringToUTF32(s: STRING): CHAR21[]
  i: INTEGER = 0;
  result: CHAR21[] = [];
  while i ≠ |s| do
    ch: CHAR21;
    if s[i] ∈ {‘uD800’ ... ‘uDBFF’} and i + 1 ≠ |s| and s[i + 1] ∈ {‘uDC00’ ... ‘uDFFF’} then
      ch = surrogatePairToSupplementaryChar(s[i], s[i + 1]);
      i = i + 2
    else
      ch = s[i];
      i = i + 1
    end if;
    result = result ⊖ [ch]
  end while;
  return result
end proc;
proc charToLowerFull(ch: CHAR21): STRING
    return ch converted to a lower case character using the Unicode full, locale-independent case mapping. A single character may be converted to multiple characters. If ch has no lower case equivalent, then the result is the string char21ToUTF16(ch).
end proc;

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

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

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

10.3 Object Utilities

10.3.1 Object Class Inquiries

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

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

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

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

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

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

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

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

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

10.3.2 Object to Boolean Conversion

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

```javascript
proc objectToBoolean(o: OBJECT): BOOLEAN
    case o of
        UNDEFINED | NULL do return false;
        BOOLEAN do return o;
        LONG | ULONG do return o.value ≠ 0;
        FLOAT32 do return o ∈ [+zero32, -zero32, NaN32];
        FLOAT64 do return o ∈ [+zero64, -zero64, NaN64];
        STRING do return o ≠ "";
        CHAR16 | NAMESPACE | COMPOUNDATTRIBUTE | CLASS | SIMPLEINSTANCE | METHODCLOSURE | DATE | REGEXP | PACKAGE do
            return true
        end case
    end proc;
```
10.3.3 Object to Primitive Conversion

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

10.3.4 Object to Number Conversions

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

```plaintext
proc objectToGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
  a: PRIMITIVEOBJECT [] objectToPrimitive(o, hintNumber, phase);
  case a of
    UDEFINED do return NaN64;
    NULL [] {false} do return +zero64;
    {true} do return 164;
    GENERALNUMBER do return a;
    STRING do return stringToFloat64(toString(a))
  end case
end proc;
```

`objectToFloat32(o, phase)` returns `o` converted to a `FLOAT32`. If `phase` is `compile`, only constant conversions are permitted.
proc objectToFloat32(o: OBJECT, phase: PHASE): FLOAT32
  a: PRIMITIVEOBJECT [] objectToPrimitive(o, hintNumber, phase);
  case a of
    UNDEFINED do return NaN1632;
    NULL [] {false} do return +zero1632;
    {true} do return 1.1632;
    GENERALNUMBER do return toFloat32(a);
    CHAR16 [] STRING do return stringToFloat32(toString(a))
  end case
end proc;

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

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

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

proc objectToExtendedInteger(o: OBJECT, phase: PHASE): EXTENDEDINTEGER
  a: PRIMITIVEOBJECT [] objectToPrimitive(o, hintNumber, phase);
  case a of
    NULL [] {false} do return 0;
    {true} do return 1;
    {undefined, NaN1632, NaN1664} do return NaN;
    {+∞1632, +∞1664} do return +∞;
    {−∞1632, −∞1664} do return −∞;
    {+zero1632, +zero1664, −zero1632, −zero1664} do return 0;
    LONG [] ULONG do return a.value;
    NONZEROFINITEFLOAT32 [] NONZEROFINITEFLOAT64 do
      r: RATIONAL [] a.value;
      if r [] INTEGER then
        throw a RangeError exception — the value a is not an integer
      end if;
      return r;
    CHAR16 [] STRING do return stringToExtendedInteger(toString(a))
  end case
end proc;

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

proc objectToInteger(o: OBJECT, phase: PHASE): INTEGER
  i: EXTENDEDINTEGER [] objectToExtendedInteger(o, phase);
  case i of
    {+∞, −∞, NaN} do throw a RangeError exception — i is not an integer;
  INTEGER do return i
end case
end proc;
proc `stringToFloat32(s: STRING): FLOAT32`
Apply the lexer grammar with the start symbol `StringNumericLiteral` to the string `s`. if the grammar cannot interpret the entire string as an expansion of `StringNumericLiteral` then
return NaN_{32}
else
  q: EXTENDEDRATIONAL
  the value of the action `Lex` applied to the obtained expansion of the nonterminal `StringNumericLiteral`;
  return extendedRationalToFloat32(q)
end if
end proc;

proc `stringToFloat64(s: STRING): FLOAT64`
Apply the lexer grammar with the start symbol `StringNumericLiteral` to the string `s`. if the grammar cannot interpret the entire string as an expansion of `StringNumericLiteral` then
return NaN_{64}
else
  q: EXTENDEDRATIONAL
  the value of the action `Lex` applied to the obtained expansion of the nonterminal `StringNumericLiteral`;
  return extendedRationalToFloat64(q)
end if
end proc;

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

10.3.5 Object to String Conversions

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

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

proc generalNumberToString(x: GENERAL_NUMBER): STRING
    case x of
        LONG ▶ ULONG do return integerToString(x.value);
        FLOAT32 do return float32ToString(x);
        FLOAT64 do return float64ToString(x)
    end case
end proc;

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

proc integerToString(i: INTEGER): STRING
    if i < 0 then return ['−'] ⊕ integerToString(−i) end if;
    q: INTEGER ▶ i/10;
    r: INTEGER ▶ i - q*10;
    c: CHAR16 ▶ integerToChar16(r + char16ToInteger('0'));
    if q = 0 then return [c] else return integerToString(q) ⊕ [c] end if
end proc;

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

proc integerToStringWithSign(i: INTEGER): STRING
    if i ≥ 0 then return ['+'] ⊕ integerToString(i)
    else return ['−'] ⊕ integerToString(−i)
    end if
end proc;

proc exponentialNotationString(digits: STRING, e: INTEGER): STRING
    mantissa: STRING;
    if |digits| = 1 then mantissa ▶ digits
    else mantissa ▶ [digits[0]] ⊕ “.” ⊕ digits[1 ...]
    end if;
    return mantissa ⊕ “e” ⊕ integerToStringWithSign(e)
end proc;

float32ToString(x) converts a FLOAT32 x to a string using fixed-point notation if the absolute value of x is between 10−6 inclusive and 10^{21} exclusive, and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a FLOAT32 value would result in the same value x (except that –ZERO_{122} would become +ZER0_{122}).
**proc** float32ToString(x: FLOAT32): STRING  
*case* x of  
- **{NaN}_{32}**  do return "NaN";  
- **{+zero}_{32}, -zero}_{32}**  do return "0";  
- **{+\infty}_{32}**  do return "Infinity";  
- **{-\infty}_{32}**  do return "-Infinity";  
  
**NONZEROFINITEFLOAT32** do  
  
  \[
  r: \text{RATIONAL} \quad \text{x.value};
  \]
  
  if \( r < 0 \) then return "-" \& float32ToString(float32Negate(x))  
  
  else  
  
  Let \( e, k, \) and \( s \) be integers such that \( k \geq 1, 10^{k-1} \leq s \leq 10^k \), \( s \lfloor 10^{e+1-k} \rfloor_{32} = x \), and \( k \) is as small as possible.  
  
  **note** \( k \) is the number of digits in the decimal representation of \( s \), \( s \) is not divisible by 10, and the least significant digit of \( s \) is not necessarily uniquely determined by the above criteria.  
  
  When there are multiple possibilities for \( s \) according to the rules above, implementations are encouraged but not required to select the one according to the following rule: Select the value of \( s \) for which \( s \lfloor 10^{e+1-k} \rfloor \) is closest in value to \( r \); if there are two such possible values of \( s \), choose the one that is even.  
  
  \[
  \text{digits: STRING} \quad \text{integerToString}(s)
  \]
  
  if \( k - 1 \leq e \leq 20 \) then return \( \text{digits} \oplus \text{repeat}(‘0’, e + 1 - k) \)  
  
  elsif \( 0 \leq e \leq 20 \) then return \( \text{digits}[0 \ldots e] \oplus “.” \oplus \text{digits}[e + 1 \ldots] \)  
  
  elsif \(-6 \leq e < 0 \) then return \( “0.” \oplus \text{repeat}(‘0’, -(e + 1)) \oplus \text{digits} \)

  else return \( \text{exponentialNotationString(digits, e)} \)

  end if  

end case  

end proc;

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

**proc** float64ToString(x: FLOAT64): STRING  
*case* x of  
- **{NaN}_{64}**  do return "NaN";  
- **{+zero}_{64}, -zero}_{64}**  do return "0";  
- **{+\infty}_{64}**  do return "Infinity";  
- **{-\infty}_{64}**  do return "-Infinity";  
  
**NONZEROFINITEFLOAT64** do  
  
  \[
  r: \text{RATIONAL} \quad \text{x.value};
  \]
  
  if \( r < 0 \) then return "-" \& float64ToString(float64Negate(x))  
  
  else  
  
  Let \( e, k, \) and \( s \) be integers such that \( k \geq 1, 10^{k-1} \leq s \leq 10^k \), \( s \lfloor 10^{e+1-k} \rfloor_{64} = x \), and \( k \) is as small as possible.  
  
  **note** \( k \) is the number of digits in the decimal representation of \( s \), \( s \) is not divisible by 10, and the least significant digit of \( s \) is not necessarily uniquely determined by the above criteria.  
  
  When there are multiple possibilities for \( s \) according to the rules above, implementations are encouraged but not required to select the one according to the following rule: Select the value of \( s \) for which \( s \lfloor 10^{e+1-k} \rfloor \) is closest in value to \( r \); if there are two such possible values of \( s \), choose the one that is even.  
  
  \[
  \text{digits: STRING} \quad \text{integerToString}(s)
  \]
  
  if \( k - 1 \leq e \leq 20 \) then return \( \text{digits} \oplus \text{repeat}(‘0’, e + 1 - k) \)  
  
  elsif \( 0 \leq e \leq 20 \) then return \( \text{digits}[0 \ldots e] \oplus “.” \oplus \text{digits}[e + 1 \ldots] \)  
  
  elsif \(-6 \leq e < 0 \) then return \( “0.” \oplus \text{repeat}(‘0’, -(e + 1)) \oplus \text{digits} \)

  else return \( \text{exponentialNotationString(digits, e)} \)

  end if  

end case  

end proc;
10.3.6 Object to Qualified Name Conversion

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

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

10.3.7 Object to Class Conversion

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

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

10.3.8 Object to Attribute Conversion

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

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

10.3.9 Implicit Coercions

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

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

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

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

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

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

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

```plaintext
proc coerceNonNull(o: OBJECT, c: CLASS): OBJECT
    result: OBJECTOPT = c.coerce(o, c);
    if result \[ none then return result
    else throw a TypeError exception — coercion failed
    end if
end proc;
```
proc coerceNonNull(o: Object, c: Class): Object
result: ObjectOpt @ c.coerce(o, c);
if result @ {none, null} then return result
else throw a TypeError exception — coercion failed
end if
end proc;

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

proc ordinaryCoerce(o: Object, c: Class): ObjectOpt
if o = null or is(o, c) then return o else return none end if
end proc;

10.3.10 Attributes

combineAttributes(a, b) returns the attribute that results from concatenating the attributes a and b.
proc combineAttributes(a: AttributeOptNotFalse, b: Attribute): Attribute

if b = false then return false
elsif a [] {none, true} then return b
elsif b = true then return a
elsif a [] Namespace then
if a = b then return a
elsif b [] Namespace then
return CompoundAttribute[namespaces: {a, b}, explicit: false, enumerable: false, dynamic: false, category: none, overrideMod: none, prototype: false, unused: false]
else return CompoundAttribute[namespaces: b.namespaces [] {a}, other fields from b]
end if
elsif b [] Namespace then
return CompoundAttribute[namespaces: a.namespaces [] {b}, other fields from a]
else
note At this point both a and b are compound attributes.
if (a.category != none and b.category != none and a.category != b.category) or (a.overrideMod != none and b.overrideMod != none and a.overrideMod != b.overrideMod) then
throw an AttributeError exception — attributes a and b have conflicting contents
else
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
end case
end proc;
10.4 Access Utilities

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

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

```
proc archetype(o: OBJECT): OBJECTOPT
    case o of
        UNDEFINED □ NULL do return none;
        BOOLEAN do return Boolean.prototype;
        LONG do return long.prototype;
        ULONG do return ulong.prototype;
        FLOAT32 do return float.prototype;
        FLOAT64 do return Number.prototype;
        CHAR16 do return char.prototype;
        STRING do return String.prototype;
        NAMESPACE do return Namespace.prototype;
        COMPOUNDATTRIBUTE do return Attribute.prototype;
        METHODCLOSURE do return Function.prototype;
        CLASS do return Class.prototype;
        SIMPLEINSTANCE □ REGEXP □ DATE □ PACKAGE do return o.archetype
    end case
end proc;
```

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

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

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

```
proc findSlot(o: OBJECT, id: INSTANCEVARIABLE): SLOT
    note o must be a SIMPLEINSTANCE or a METHODCLOSURE in order to have slots.
    matchingSlots: SLOT { } □ {s | s □ o.slots such that s.id = id};
    return the one element of matchingSlots
end proc;
```

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

```
proc setupVariable(v: VARIABLE)
    setup: () □ CLASSOPT □ {none, busy} □ v.setup;
    case setup of
        () □ CLASSOPT do
            v.setup □ busy;
            type: CLASSOPT □ setup();
            if type = none then type □ Object end if;
            v.type □ type;
            v.setup □ none;
        {none} do nothing;
        {busy} do
            throw a ConstantError exception — a constant’s type or initialiser cannot depend on the value of that constant
        end case
end proc;
```
writeVariable(v, newValue, clearInitializer) writes the value newValue into the mutable or immutable variable v. newValue is coerced to v’s type. If the clearInitializer flag is set, then the caller has just evaluated v’s initialiser and is supplying its result in newValue. In this case writeVariable atomically clears v.initializer while writing v.value. In all other cases the presence of an initialiser or an existing value will prevent an immutable variable’s value from being written.

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

10.5 Environmental Utilities

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

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

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

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

getRegionalEnvironment(env) returns all frames in env up to and including the first regional frame. A regional frame is either any frame other than a with frame or local block frame, a local block frame directly enclosed in a class, or a local block frame directly enclosed in a with frame directly enclosed in a class.

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

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

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

getPackageFrame(env) returns the innermost package frame in env.
proc getPackageFrame(env: Environment): Package
i: INTEGER □ 0;
while env[i] □ PACKAGE do i ▷ i + 1 end while;
note Every environment ends with a PACKAGE frame, so one will always be found.
return env[i]
end proc;

10.6 Property Lookup

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

proc findLocalSingletonProperty(o: NonWithFrame □ SimpleInstance □ RegExp □ Date, multiname: Multiname, access: Access): SingletonPropertyOpt
matchingLocalBindings: LocalBinding[] □ {b | b □ a.localBindings such that
b.qname □ multiname and accessesOverlap(b.accesses, access)};
note If the same property was found via several different bindings b, then it will appear only once in the set matchingProperties.
matchingProperties: SingletonProperty {} □ {b.content | b □ matchingLocalBindings};
if matchingProperties = {} then return none
elsif |matchingProperties| = 1 then return the one element of matchingProperties
else
throw a ReferenceError exception — this access is ambiguous because the bindings it found belong to several different local properties
end if
end proc;

instancePropertyAccesses(m) returns instance property’s ACCESSSET.
proc instancePropertyAccesses(m: InstanceProperty): ACCESSSET
case m of
  InstanceVariable □ InstanceMethod do return readWrite;
  InstanceGetter do return read;
  InstanceSetter do return write
end case
end proc;

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

proc findLocalInstanceProperty(c: Class, multiname: Multiname, accesses: AccessSet): InstancePropertyOpt
matches: InstanceProperty {} □ {m | m □ c.instanceProperties such that m.multiname □ multiname ≠ {} and accessesOverlap(instancePropertyAccesses(m), accesses)};
if matches = {} then return none
elsif |matches| = 1 then return the one element of matches
else
throw a ReferenceError exception — this access is ambiguous because it found several different instance properties in the same class
end if
end proc;

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

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

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

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

proc findBaseInstanceProperty(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
  note Start from the root class (Object) and proceed through more specific classes that are ancestors of c.
for each s □ ancestors(c) do
  m: INSTANCEPROPERTYOPT □ findLocalInstanceProperty(s, multiname, accesses);
  if m ≠ none then return m end if
end for each;
return none
end proc;

getDerivedInstanceProperty(c, mBase, accesses) returns the most derived instance property whose name includes that of mBase and whose accesses that have a nonempty intersection with accesses. The caller of getDerivedInstanceProperty ensures that such an instance property always exists. If accesses is readWrite then it is possible that this search could find both a getter and a setter defined in the same class; in this case either the getter or the setter is returned at the implementation’s discretion.
proc getDerivedInstanceProperty(c: CLASS, mBase: INSTANCEPROPERTY, accesses: ACCESSSET): INSTANCEPROPERTY
if some m \in c.instanceProperties satisfies mBase.multiname \notin m.multiname and
 accessesOverlap(instancePropertyAccesses(m), accesses) then
   return m
else return getDerivedInstanceProperty(c.super, mBase, accesses)
end if
end proc;

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

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

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

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

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

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

10.7 Reading

If r is an OBJECT, readReference(r, phase) returns it unchanged. If r is a REFERENCE, readReference reads r and returns the result. If phase is compile, only constant expressions can be evaluated in the process of reading r.
proc readReference(r: ObjRef, phase: Phase): Object
result: ObjectOpt;
case r of
  Object do result [] r;
  LexicalReference do result [] lexicalRead(r.env, r.variableMultiname, phase);
  DotReference do
    result [] r.limit.read(r.base, r.limit, r.multiname, none, true, phase);
  BracketReference do
    result [] r.limit.bracketRead(r.base, r.limit, r.args, true, phase)
end case;
if result ≠ none then return result
else
  throw a ReferenceError exception — property not found, and no default value is available
end if
end proc;

dotRead(o, Multiname, phase) reads and returns the value of the Multiname property of o. dotRead throws an error if the property does not exist and no default value was available for it.
proc dotRead(o: Object,_multiname: Multiname, phase: Phase): Object
  limit: Class [] objectType(o);
  result: ObjectOpt [] limit.read(o, limit, Multiname, none, true, phase);
if result = none then
  throw a ReferenceError exception — property not found, and no default value is available
end if;
return result
end proc;

readLength(o, phase) reads and returns the value of the length property of o, ensuring that it is an integer between 0 and arrayLimit inclusive.
proc readLength(o: Object, phase: Phase): Integer
  value: Object [] dotRead(o, {public::"length"}, phase);
if value [] GeneralNumber then throw a TypeError exception — length not an integer
end if;
  limit: IntegerOpt [] checkInteger(value);
if length = none then throw a RangeError exception — length not an integer
elsif 0 ≤ length ≤ arrayLimit then return length
else throw a RangeError exception — length out of range
end if
end proc;

indexRead(o, i, phase) returns the value of o[i] or none if no such property was found; unlike dotRead, indexRead does not return a default value for missing properties. i should always be a valid array index.
proc indexRead(o: Object, i: Integer, phase: Phase): ObjectOpt
  note 0 ≤ i < arrayLimit;
  limit: Class [] objectType(o);
  x: Float64 [] iso;
  result: ObjectOpt [] limit.bracketRead(o, limit, [x], false, phase);
if result ≠ none and not hasProperty(o, x, true, phase) then
  At the implementation’s discretion either do nothing, set result to none, or throw a ReferenceError.
end if;
return result
end proc;

ordinaryBracketRead(o, limit, args, undefinedIfMissing, phase) evaluates the expression o[args] when o is a native object. Host objects may either also use ordinaryBracketRead or choose a different procedure P to evaluate o[args] by writing P into objectType(o).bracketRead.
limit is used to handle the expression `super(o)[args]`, in which case limit is the superclass of the class inside which the super expression appears. Otherwise, limit is set to `objectType(o)`.

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

```
proc lexicalRead(env: ENVIRONMENT, multiname: MULTINAME, phase: PHASE): OBJECT
  i: INTEGER = 0;
  while i < |env| do
    frame: FRAME = env[i];
    result: OBJECTOPT = none;
    case frame of
      PACKAGE do
        limit: CLASS = objectType(frame);
        result = limit.read(frame, limit, multiname, env, false, phase);
      end case;
      LOCALFRAME do
        m: SINGLETONPROPERTYOPT = findLocalSingletonProperty(frame, multiname, read);
        if m ≠ none then result = readSingletonProperty(m, phase) end if;
      end case;
      WITHFRAME do
        value: OBJECTOPT = frame.value;
        if value = none then
          case phase of
            {compile} do
              throw a ConstantError exception — cannot read a with statement’s frame from a constant expression;
            {run} do
              throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s expression has been evaluated
            end case
          end if;
        end case;
        limit: CLASS = objectType(value);
        result = limit.read(value, limit, multiname, env, false, phase)
      end case;
      if result ≠ none then return result end if;
    i = i + 1
  end while;
  throw a ReferenceError exception — no property found with the name multiname
end proc;
```
mBase: INSTANCEPROPERTYOPT  findBaseInstanceProperty(limit, multiname, read);
if mBase ≠ none then return readInstanceProperty(o, limit, mBase, phase) end if;
if limit ≠ objectType(o) then return none end if;

default: BOOLEAN  env ≠ none and o Δ CLASS;
m: PROPERTYOPT  findArchetypeProperty(o, multiname, read, flat);
case m of
{none} do
  if undefinedIfMissing and o Δ SIMPLEINSTANCE Δ DATE Δ REGEXP Δ PACKAGE and not o.sealed then
  case phase of
  {compile} do
    throw a ConstantError exception — a constant expression cannot read dynamic properties;
  {run} do return undefined
  end case
  else return none
  end if;
SINGLETONPROPERTY do return readSingletonProperty(m, phase);
INSTANCEPROPERTY do
  if o Δ CLASS or env = none then
    throw a ReferenceError exception — cannot read an instance property without supplying an instance
  end if;
  this: OBJECT  readImplicitThis(env);
  return readInstanceProperty(this, objectType(this), m, phase)
end case
end proc;

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

proc readInstanceSlot(o: OBJECT, qname: QUALIFIEDNAME, phase: PHASE): OBJECT
c: CLASS  objectType(o);
mBase: INSTANCEPROPERTYOPT  findBaseInstanceProperty(c, {qname}, read);
ote readInstanceProperty is only called in cases where the instance property is known to exist, so mBase cannot be none here.
return readInstanceProperty(o, c, mBase, phase)
end proc;
proc readInstanceProperty(\textit{this}: \textbf{Object}, \textit{c}: \textbf{Class}, \textit{mBase}: \textbf{InstanceProperty}, \textit{phase}: \textbf{Phase}): \textbf{Object} \\
\textbf{case} \textit{m} \textbf{of} \\
\quad \textbf{INSTANCE VARIABLE} \textbf{do} \\
\quad \quad \textbf{if} \: \textit{phase} = \textbf{compile} \: \textbf{and} \: \textbf{not} \: \textit{m}.\textbf{immutable} \: \textbf{then} \\
\quad \quad \quad \textbf{throw} \: \textbf{a} \: \textbf{ConstantError} \: \textbf{exception} — \: \textbf{a} \: \textbf{constant} \: \textbf{expression} \: \textbf{cannot} \: \textbf{read} \: \textbf{mutable} \: \textbf{variables} \\
\quad \textbf{end} \: \textbf{if}; \\
\quad \textit{v}: \textbf{ObjectOpt} \: \textbf{ InterruptedException} \:\textbf{findSlot}(\textit{this}, \textit{m}).\textbf{value}; \\
\quad \textbf{if} \: \textit{v} = \textbf{none} \: \textbf{then} \\
\quad \quad \textbf{case} \: \textit{phase} \: \textbf{of} \\
\quad \quad \quad \{ \textbf{compile} \} \: \textbf{do} \\
\quad \quad \quad \quad \textbf{throw} \: \textbf{a} \: \textbf{ConstantError} \: \textbf{exception} — \: \textbf{cannot} \: \textbf{read} \:\textbf{uninitialised} \: \textbf{const} \: \textbf{variables} \: \textbf{from} \: \textbf{a} \: \textbf{constant} \: \textbf{expression}; \\
\quad \quad \quad \{ \textbf{run} \} \: \textbf{do} \\
\quad \quad \quad \quad \textbf{throw} \: \textbf{an} \: \textbf{UninitializedError} \: \textbf{exception} — \: \textbf{cannot} \: \textbf{read} \: \textbf{a} \: \textbf{const} \: \textbf{instance} \: \textbf{variable} \: \textbf{before} \: \textbf{it} \: \textbf{is} \: \textbf{initialised} \\
\quad \quad \textbf{end} \: \textbf{case} \\
\textbf{end} \: \textbf{if}; \\
\quad \textbf{return} \: \textit{v}; \\
\quad \textbf{INSTANCE METHOD} \: \textbf{do} \\
\quad \quad \textit{slots}: \textbf{Slot} \{\} \: \{ \textbf{new} \: \textbf{Slot}:: \: \textbf{iVarFunctionLength}, \: \textbf{value}: (\textit{m}.\textbf{length})_64\}; \\
\quad \textbf{return} \: \textbf{MethodClosure}(\textit{this}, \: \textit{method}: \: \textit{m}, \: \textit{slots}: \: \textit{slots}); \\
\quad \textbf{INSTANCE GETTER} \: \textbf{do} \: \textbf{return} \: \textit{m}.\textbf{call}(\textit{this}, \: \textit{phase}); \\
\quad \textbf{INSTANCE SETTER} \: \textbf{do} \\
\quad \quad \textit{m} \: \textbf{cannot} \: \textbf{be} \: \textbf{an} \: \textbf{INSTANCE SETTER} \: \textbf{because} \: \textbf{these} \: \textbf{are} \: \textbf{only} \: \textbf{represented} \: \textbf{as} \: \textbf{write-only} \: \textbf{properties}. \\
\textbf{end} \: \textbf{case} \\
\textbf{end} \: \textbf{proc};
proc readSingletonProperty(m: SINGLETONPROPERTY, phase: PHASE): OBJECT
  case m of
    {forbidden} do
      throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if any block inside the current region shadows it;
  DYNAMIC_VAR do
    if phase = compile then
      throw a ConstantError exception — a constant expression cannot read mutable variables
    end if;
    value: OBJECT □ UNINSTANTIATEDFUNCTION □ m.value;
    note value can be an UNINSTANTIATEDFUNCTION only during the compile phase, which was ruled out above.
    return value;
  VARIABLE do
    if phase = compile and not m.immutable then
      throw a ConstantError exception — a constant expression cannot read mutable variables
    end if;
    value: VARIABLEVALUE □ m.value;
    case value of
      OBJECT do return value;
      {none} do
        if not m.immutable then throw an UninitializedError exception end if;
        note Try to run a const variable’s initialiser if there is one.
        Evaluate setupVariable(m) and ignore its result;
        initializer: INITIALIZER □ {none, busy} □ m.initializer;
        if initializer □ {none, busy} then
          case phase of
            {compile} do
              throw a ConstantError exception — a constant expression cannot access a constant with a missing or recursive initialiser;
            end case
          end if;
        m.initializer □ busy;
        coercedValue: OBJECT;
        try
          newValue: OBJECT □ initializer(m.initializerEnv, compile);
          coercedValue □ writeVariable(m, newValue, true)
        catch x: SEMANTICEXCEPTION do
          note If initialisation failed, restore m.initializer to its original value so it can be tried later.
          m.initializer □ initializer;
          throw x
        end try;
        return coercedValue;
      UNINSTANTIATEDFUNCTION do
        note An uninstantiated function can only be found when phase = compile.
        throw a ConstantError exception — an uninstantiated function is not a constant expression
      end case;
  GETTER do
    env: ENVIRONMENTOPT □ m.env;
    if env = none then
      note An uninstantiated getter can only be found when phase = compile.
      throw a ConstantError exception — an uninstantiated getter is not a constant expression
    end if;
    return m.call(env, phase);
  SETTER do
    m cannot be a SETTER because these are only represented as write-only properties.
  end case
end proc;

10.8 Writing

If \( r \) is a reference, \( \text{writeReference}(r, \text{newValue}) \) writes \( \text{newValue} \) into \( r \). An error occurs if \( r \) is not a reference. \( \text{writeReference} \) is never called from a constant expression.

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

\( \text{dotWrite}(o, \text{multiname, newValue, phase}) \) is a simplified interface to write \( \text{newValue} \) into the \( \text{multiname} \) property of \( o \).

\[
\text{proc dotWrite}(o: \text{OBJECT, multiname: MULTINAME, newValue: OBJECT, phase: \{run\}}) \\
\text{limit: CLASS \{ objectType(o); \\
\text{result: \{none, ok\}} \[ limit.write(o, limit, multiname, none, newValue, true, phase); \\
\text{if result} = \text{none then} \\
\quad \text{throw a ReferenceError exception — property not found and could not be created} \\
end if \\
end proc;
\]

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

\[
\text{proc writeLength(o: \text{OBJECT, length: INTEGER, phase: \{run\}}) \\
\quad \text{if length} < 0 \text{ or length} > \text{arrayLimit then} \\
\quad \quad \text{throw a RangeError exception — length out of range} \\
\quad \text{end if}; \\
\quad \text{Evaluate dotWrite(o, \{public::"length"\}, length\text{\#4}, phase) and ignore its result} \\
end proc;
\]
proc indexWrite(o: OBJECT, i: INTEGER, newValue: OBJECTOpt, phase: {run})
if i < 0 or i ≥ arrayLimit then throw a RangeError exception — index out of range
end if;
limit: CLASS [] objectType(o);
if newValue = none then
  deleteResult: BOOLEANOpt [] limit.bracketDelete(o, limit, [i64], phase);
  if deleteResult = false then
    throw a ReferenceError exception — cannot delete element
  end if
else
  writeResult: {none, ok} [] limit.bracketWrite(o, limit, [i64], newValue, true, phase);
  if writeResult = none then
    throw a ReferenceError exception — element not found and could not be created
  end if
end if
end proc;

proc ordinaryBracketWrite(o: OBJECT, limit: CLASS, args: OBJECT[], newValue: OBJECT, createIfMissing: BOOLEAN, phase: {run}): {none, ok}
if |args| ≠ 1 then
  throw an ArgumentError exception — exactly one argument must be supplied
end if;
qname: QUALIFIEDNAME [] objectToQualifiedName(args[0], phase);
return limit.write(o, limit, {qname}, none, newValue, createIfMissing, phase)
end proc;
proc lexicalWrite(env: ENVIRONMENT, multiname: MULTINAME, newValue: OBJECT, createIfMissing: BOOLEAN, phase: \{run\})
  i: INTEGER = 0;
  while i < |env|
do
    frame: FRAME = env[i];
    result: \{none, ok\} = none;
    case frame of
      PACKAGE -> CLASS do
        limit: CLASS = objectType(frame);
        result = limit.write(frame, limit, multiname, env, newValue, false, phase);
      end
      PARAMETERFRAME -> LOCALFRAME do
        m: SINGLETONPROPERTYOPT = findLocalSingletonProperty(frame, multiname, write);
        if m ≠ none then
          Evaluate writeSingletonProperty(m, newValue, phase) and ignore its result;
          result = ok
        end if;
      end
      WITHFRAME do
        value: OBJECTOPT = frame.value;
        if value = none then
          throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s expression has been evaluated
        end if;
        limit: CLASS = objectType(value);
        result = limit.write(value, limit, multiname, env, newValue, false, phase);
      end case;
      if result = ok then return end if;
      i = i + 1
    end while;
  if createIfMissing then
    pkg: PACKAGE = getPackageFrame(env);
    note Try to write the variable into pkg again, this time allowing new dynamic bindings to be created dynamically.
    limit: CLASS = objectType(pkg);
    result: \{none, ok\} = limit.write(pkg, limit, multiname, env, newValue, true, phase);
    if result = ok then return end if
  end if;
  throw a ReferenceError exception — no existing property found with the name multiname and one could not be created
end proc;
The caller must make sure that the created property does not already exist and does not conflict with any other property.

```
proc ordinaryWrite(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOpt, newValue: OBJECT, createIfMissing: BOOLEAN, phase: {run}): {none, ok}
mBase: INSTANCEPROPERTYOpt [] findBaseInstanceProperty(limit, multiname, write);
if mBase ≠ none then
    Evaluate writeInstanceProperty(o, limit, mBase, newValue, phase) and ignore its result;
    return ok
end if;
if limit ≠ objectType(o) then
    return none
end if;
m: PROPERTYOPT [] findArchetypeProperty(o, multiname, write, true);
case m of
    {none} do
        if createIfMissing and o SIMPLEINSTANCE » DATE » REGEXP » PACKAGE and not o.sealed and
            (some qname [] multiname satisfies qname.namespace = public) then
            note Before trying to create a new dynamic property named qname, check that there is no read-only fixed
            property with the same name.
            if findBaseInstanceProperty(objectType(o), {qname}, read) = none and
                findArchetypeProperty(o, {qname}, read, true) = none then
                Evaluate createDynamicProperty(o, qname, false, true, newValue) and ignore its result;
                return ok
            end if
        end if
        return none;
    SINGLETONPROPERTY do
        Evaluate writeSingletonProperty(m, newValue, phase) and ignore its result;
        return ok;
    INSTANCEPROPERTY do
        if o CLASS or env = none then
            throw a ReferenceError exception — cannot write an instance property without supplying an instance
        end if;
        this: OBJECT [] readImplicitThis(env);
        Evaluate writeInstanceProperty(this, objectType(this), m, newValue, phase) and ignore its result;
        return ok
    end case
end proc;
```
proc writeInstanceProperty(this: Object, c: Class, mBase: InstanceProperty, newValue: Object, phase: {run})

m: InstanceProperty □ getDerivedInstanceProperty(c, mBase, write);

case m of

  InstanceVariable do
    s: Slot □ findSlot(this, m);
    coercedValue: Object □ coerce(newValue, m.type);
    if m.immutable and s.value ≠ none then
      throw a ReferenceError exception — cannot initialise a const instance variable twice
    end if;
    s.value □ coercedValue;
  
  InstanceMethod do
    m cannot be an InstanceGetter because these are only represented as read-only properties.
  
  InstanceGetter do Evaluate m.call(this, newValue, phase) and ignore its result

  end case
end proc;

proc writeSingletonProperty(m: SingletonProperty, newValue: Object, phase: {run})

case m of

  {forbidden} do
    throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if
    any block inside the current region shadows it;
  
  Variable do Evaluate writeVariable(m, newValue, false) and ignore its result;
  
  DynamicVar do m.value □ newValue;
  
  Getter do
    m cannot be a Getter because these are only represented as read-only properties.
  
  Setter do
    env: EnvironmentOpt □ m.env;
    note All instances are resolved for the run phase, so env ≠ none.
    Evaluate m.call(newValue, env, phase) and ignore its result

  end case
end proc;

10.9 Deleting

If r is a reference, deleteReference(r) deletes it. If r is an object, this function signals an error in strict mode or returns true in non-strict mode. deleteReference is never called from a constant expression.

proc deleteReference(r: ObjOrRef, strict: Boolean, phase: {run}): BOOLEAN

result: BooleanOpt;

case r of

  Object do
    if strict then
      throw a ReferenceError exception — a non-reference is not a valid target for delete in strict mode
    else result □ true
    end if;
  
  LexicalReference do result □ lexicalDelete(r.env, r.variableMultiname, phase);
  
  DotReference do
    result □ r.limit.delete(r.base, r.limit, r.multiname, none, phase);
  
  BracketReference do
    result □ r.limit.bracketDelete(r.base, r.limit, r.args, phase)

  end case;

  if result ≠ none then return result else return true end if

end proc;
proc ordinaryBracketDelete(o: OBJECT, limit: CLASS, args: OBJECT[], phase: {run}): BOOLEANOPT
if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
end if;
quname: QUALIFIEDNAME = objectToQualifiedName(args[0], phase);
return limit.delete(o, limit, {qname}, none, phase)
end proc;

proc lexicalDelete(env: ENVIRONMENT, multiname: MULTINAME, phase: {run}): BOOLEAN
i: INTEGER = 0;
while i < |env| do
    frame: FRAME = env[i];
    result: BOOLEANOPT = none;
    case frame of
        PACKAGE » CLASS do
            limit: CLASS = objectType(frame);
            result = limit.delete(frame, limit, multiname, env, phase);
        PARAMETERFRAME » LOCALFRAME do
            if findLocalSingletonProperty(frame, multiname, write) ≠ none then
                result = false
            end if;
        end if;
        WITHFRAME do
            value: OBJECTOPT = frame.value;
            if value = none then
                throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s expression has been evaluated
            end if;
            limit: CLASS = objectType(value);
            result = limit.delete(value, limit, multiname, env, phase)
        end case;
    end case;
    if result ≠ none then return result end if;
    i = i + 1
end while;
return true
end proc;
proc ordinaryDelete(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, phase: {run}):
  BOOLEAN
if findBaseInstanceProperty(limit, multiname, write) ≠ none then return false end if;
if limit ≠ objectType(o) then return none end if;
m: PROPERTYOPT findArchetypeProperty(o, multiname, write, true);
case m of
  {none} do return none;
  {forbidden} do
    throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if any block inside the current region shadows it;
  VARIABLE » GETTER » SETTER do return false;
  DYNAMICVAR do
    if m.sealed then return false
    else
      o.localBindings {b | b.sharedWith o.localBindings such that b.qname œ multiname or b.content ≠ m};
      return true
    end if;
  INSTANCEPROPERTY do
    if o.CLASS or env = none then return false end if;
    Evaluate readImplicitThis(env) and ignore its result;
    return false
  end case
end proc;

10.10 Enumerating

proc ordinaryEnumerate(o: OBJECT): OBJECT{}
  e1: OBJECT{} ≡ enumerateInstanceProperties(objectType(o));
  e2: OBJECT{} ≡ enumerateArchetypeProperties(o);
  return e1 » e2
end proc;

proc enumerateInstanceProperties(c: CLASS): OBJECT{}
  e: OBJECT{} ≡ {};
  for each m œ c.instanceProperties do
    if m.enumerable then
      e ≡ e » {qname.id | qname œ m.multiname such that qname.namespace = public}
    end if
  end for each;
  super: CLASSOPT ≡ c.super;
  if super = none then return e
  else return e » enumerateInstanceProperties(super)
  end if
end proc;

proc enumerateArchetypeProperties(o: OBJECT): OBJECT{}
  e: OBJECT{} ≡ {};
  for each a œ {o} » archetypes(o) do
    if a » BINDINGOBJECT then e ≡ e » enumerateSingletonProperties(a) end if
  end for each;
  return e
end proc;
proc enumerateSingletonProperties(o: BindingObject): Object{}
e: Object{} ∈ []; for each b ∈ o.localBindings do
  if b.enumerable and b.qname.namespace = public then e ∈ {b.qname.id} end if
end for each;
if o ∈ CLASS then
  super: CLASSOpt = o.super;
  if super ≠ none then e ∈ enumerateSingletonProperties(super) end if
end if;
return e end proc;

10.11 Calling Instances

proc call(this: Object, a: Object, args: Object[], phase: PHASE): Object
  case a of
    UNDEFINED ≠ NULL ≠ BOOLEAN ≠ GENERALNUMBER ≠ CHAR16 ≠ STRING ≠ NAMESPACE ≠
    COMPOUNDATEtribute ≠ DATE ≠ REGEXP ≠ PACKAGE do
      throw a TypeError exception;
    CLASS do return a.call(this, a, args, phase);
    SIMPLEINSTANCE do
      f: (OBJECT ≠ SIMPLEINSTANCE ≠ OBJECT[] ≠ PHASE ≠ OBJECT) ≠ {none} ≠ a.call;
      if f ≠ none then throw a TypeError exception end if;
      return f(this, a, args, phase);
    METHODCLOSURE do
      m: INSTANCEMETHOD ≠ a.method;
      return m.call(a.this, args, phase)
  end case
end proc;

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

proc sameAsConstruct(this: Object, c: CLASS, args: Object[], phase: PHASE): Object
  return construct(c, args, phase)
end proc;
10.12 Creating Instances

proc construct(a: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  case a of
    undefined | null | boolean | genericnumber | char16 | string | namespace |
      compoundattribute | methodclosure | date | regexp | package do
    throw a TypeError exception;
    class do return a.construct(a, args, phase);
    simpleinstance do
      f: (simpleinstance [object] [object] [phase] object) [none] [none] a.construct;
      if f = none then throw a TypeError exception end if;
      return f(a, args, phase)
    end case
  end proc;

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

proc createSimpleInstance(c: class, archetype: objectopt,
   call: (object [simpleinstance] [object] [object] [phase] object) [none],
   construct: (simpleinstance [object] [object] [phase] object) [none], env: environmentopt):
  simpleinstance
  slots: slot {;};
  for each s :: ancestors(c) do
    for each m :: s.instanceProperties do
      if m :: instancevariable then
        slot: slot [new slot] id: m, value: m.defaultValue[]
        slots [slots] [slot]
      end if
    end for each
  end for each;
  return new simpleinstance [localbindings: {}], archetype: archetype, sealed: not c.dynamic, type: c, slots: slots,
  call: call, construct: construct, env: env[]
end proc;

proc callInit(this: simpleinstance, c: classopt, args: object[], phase: {run})
  init: (simpleinstance [object] [run] [run] [none]) [none] [none];
  if c = none then init [] c.init end if;
  if init = none then evaluate init(this, args, phase) and ignore its result
  else
    if args = [] then
      throw an ArgumentError exception — the default constructor does not take any arguments
    end if
  end if
end proc;
10.13 Adding Local Definitions

```plaintext
proc defineSingletonProperty(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE { },
overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, accesses: ACCESSSET, m: SINGLETONPROPERTY):
MULTINAME
innerFrame: NonWITHFRAME [] env[0];
if overrideMod ≠ none then
    throw an AttributeError exception — a local definition cannot have the override attribute
end if;
if explicit and innerFrame [] PACKAGE then
    throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
end if;
namespaces2: NAMESPACE { } [] namespaces;
if namespaces2 = {} then namespaces2 [] {public} end if;
multiname: MULTINAME { ns:id | [] ns [] namespaces2};
regionalEnv: FRAME [] getRegionalEnvironment(env);
if some b [] innerFrame.localBindings satisfies
    b.qname [] multiname and accessesOverlap(b.accesses, accesses) then
    throw a DefinitionError exception — duplicate definition in the same scope
end if;
if innerFrame [] CLASS and id = innerFrame.name then
    throw a DefinitionError exception — a static property of a class cannot have the same name as the class,
    regardless of the namespace
end if;
for each frame [] regionalEnv[1 ...] do
    if frame [] WITHFRAME and (some b [] frame.localBindings satisfies b.qname [] multiname and
    accessesOverlap(b.accesses, accesses) and b.content ≠ forbidden) then
        throw a DefinitionError exception — this definition would shadow a property defined in an outer scope within
        the same region
    end if;
end for each;
newBindings: LOCALBINDING { } [] {LOCALBINDING[]name: qname, accesses: accesses, explicit: explicit,
    enumerable: true, content: m[] [] qname [] multiname};
innerFrame.localBindings [] innerFrame.localBindings [] newBindings;
```

**note** Mark the bindings of multiname as forbidden in all non-innermost frames in the current region if they haven’t
been marked as such already.

```plaintext
newForbiddenBindings: LOCALBINDING { } [] {LOCALBINDING[]name: qname, accesses: accesses, explicit: true,
    enumerable: true, content: forbidden[] [] qname [] multiname};
```

for each frame [] regionalEnv[1 ...] do
    **note** Since frame [] CLASS here, a CLASS frame never gets a forbidden binding.
    if frame [] WITHFRAME then
        frame.localBindings [] frame.localBindings [] newForbiddenBindings
    end if
end for each;
return multiname
end proc;
```

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

proc searchForOverrides(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
mBase: INSTANCEPROPERTYOPT " none;
s: CLASSOPT " c.super;
if s " none then
  for each qname " multiname do
    m: INSTANCEPROPERTYOPT " findBaseInstanceProperty(s, {qname}, accesses);
    if mBase = none then mBase " m
    elsif m " none and m " mBase then
      throw a DefinitionError exception — cannot override two separate superclass methods at the same time
    end if
  end for each
end if;
return mBase
end proc;
proc defineInstanceProperty(c: CLASS, ctx: CONTEXT, id: STRING, namespaces: NAMESPACE[]);

    overrideMod: OVERRIDE_MODIFIER, explicit: BOOLEAN, m: INSTANCEPROPERTY): INSTANCEPROPERTY;

    if explicit then
        throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
    end if;

    accesses: ACCESSSET [] instancePropertyAccesses(m);
    requestedMultiname: MULTINAME [ ns: id | ns namespaces ];
    openMultiname: MULTINAME [ ns: id | ns ctx.openNamespaces ];
    definedMultiname: MULTINAME;
    searchedMultiname: MULTINAME;

    if requestedMultiname = {} then
        definedMultiname [] searchedMultiname; searchedMultiname [] requestedMultiname
    end if;

    mBase: INSTANCEPROPERTY [] searchForOverrides(c, searchedMultiname, accesses);
    mOverridden: INSTANCEPROPERTY [] none;

    if mBase ≠ none then
        mOverridden [] getDerivedInstanceProperty(c, mBase, accesses);
        definedMultiname [] mOverridden.multiname;

        if not (requestedMultiname [] definedMultiname) then
            throw a DefinitionError exception — cannot extend the set of a property’s namespaces when overriding it
        end if;

        goodKind: BOOLEAN;

        case m of
            INSTANCE_VARIABLE do goodKind [] mOverridden [] INSTANCE_VARIABLE;
            INSTANCE_GETTER do
                goodKind [] mOverridden [] INSTANCE_VARIABLE [] INSTANCE_GETTER;
            INSTANCE_SETTER do
                goodKind [] mOverridden [] INSTANCE_VARIABLE [] INSTANCE_SETTER;
            INSTANCE_METHOD do goodKind [] mOverridden [] INSTANCE_METHOD
        end case;

        if not goodKind then
            throw a DefinitionError exception — a method can override only another method, a variable can override only another variable, a getter can override only a getter or a variable, and a setter can override only a setter or a variable
        end if;

        if mOverridden.final then
            throw a DefinitionError exception — cannot override a final property
        end if;

    end if;

    if some m2 [] c.instanceProperties satisfies m2.multiname [] definedMultiname ≠ {} and
        accessesOverlap(instancePropertyAccesses(m2), accesses) then
        throw a DefinitionError exception — duplicate definition in the same scope
    end if;

    case overrideMod of
        { none } do
            if mBase ≠ none then
                throw a DefinitionError exception — a definition that overrides a superclass’s property must be marked with the override attribute
            end if;

            if searchForOverrides(c, openMultiname, accesses) ≠ none then
                throw a DefinitionError exception — this definition is hidden by one in a superclass when accessed without a namespace qualifier; in the rare cases where this is intentional, use the override(false) attribute
            end if;
        end if;

        { false } do
            
        end if;

        }
if \( mBase \neq \text{none} \) then
  throw a \texttt{DefinitionError} exception — this definition is marked with \texttt{override(false)} but it overrides a superclass’s property
end if;
{true} do
  if \( mBase = \text{none} \) then
    throw a \texttt{DefinitionError} exception — this definition is marked with \texttt{override} or \texttt{override(true)} but it doesn’t override a superclass’s property
  end if;
  {undefined} do nothing
end case;
\( m\).multiname \[ \text{definedMultiname} \);
\( c\).instanceProperties \[ \text{c.instanceProperties} \[ \{m\} \];
return \( m\)\texttt{Overridden}
end proc;

10.15 Instantiation

\texttt{proc instantiateFunction(uf: UNINSTANTIATEDFUNCTION, env: ENVIRONMENT): SIMPLEINSTANCE}\n\( c\): \texttt{CLASS} \[ uf\).type;\n\( i\): \texttt{SIMPLEINSTANCE} \[ \text{createSimpleInstance}(c, c\).prototype, uf\).call, uf\).construct, env);\nEvaluate \texttt{dotWrite}(i, \{public::“length”\}, (uf\).length)\[64, \text{run} \) and ignore its result;
if \( c = \text{PrototypeFunction} \) then
  prototype: \texttt{OBJECT} \[ \text{construct}(Object, [], \text{run});\n  Evaluate \texttt{dotWrite}(prototype, \{public::“constructor”\}, i, \text{run}) and ignore its result;
  Evaluate \texttt{dotWrite}(i, \{public::“prototype”\}, prototype, \text{run}) and ignore its result
end if;
\texttt{instantiations: SIMPLEINSTANCE}\[\{\} \[ uf\).instantiations;\nif \( \text{instantiations} \neq \{\} \) then
  Suppose that \textit{instantiateFunction} were to choose at its discretion some element \( i2 \) of \textit{instantiations}, assign \( i2\).env \[ env, and return \( i \). If the behaviour of doing that assignment were observationally indistinguishable by the rest of the program from the behaviour of returning \( i \) without modifying \( i2\).env, then the implementation may, but does not have to, return \( i2 \) now, discarding (or not even bothering to create) the value of \( i \).

\textbf{note}  The above rule allows an implementation to avoid creating a fresh closure each time a local function is instantiated if it can show that the closures would behave identically. This optimisation is not transparent to the programmer because the instantiations will be \texttt{===} to each other and share one set of properties (including the \texttt{prototype} property, if applicable) rather than each having its own. ECMAScript programs should not rely on this distinction.
end if;
uf\).instantiations \[ \text{instantiations} \[ \{i\};\nreturn \( i \);
end proc;
proc instantiateProperty(m: SINGLETONPROPERTY, env: ENVIRONMENT): SINGLETONPROPERTY  
  case m of  
    {forbidden} do return m;  
    VARIABLE do  
      note m.setup = none because Setup must have been called on a frame before that frame can be instantiated.  
      value: VARIABLEValue [] m.value;  
      if value [] UNINSTANTIATEDFUNCTION then  
        value [] instantiateFunction(value, env)  
      end if;  
      return new VARIABLE[type: m.type, value: value, immutable: m.immutable, setup: none,  
                      initializer: m.initializer, initializerEnv: env]  
    DYNAMICVAR do  
      value: OBJECT [] UNINSTANTIATEDFUNCTION [] m.value;  
      if value [] UNINSTANTIATEDFUNCTION then  
        value [] instantiateFunction(value, env)  
      end if;  
      return new DYNAMICVAR[value: value, sealed: m.sealed]  
    GETTER do  
      case m.env of  
        ENVIRONMENT do return m;  
        {none} do return new GETTER[call: m.call, env: env]  
      end case;  
    SETTER do  
      case m.env of  
        ENVIRONMENT do return m;  
        {none} do return new SETTER[call: m.call, env: env]  
      end case  
    end case  
  end proc;  

tuple PROPERTYTRANSLATION  
  from: SINGLETONPROPERTY,  
  to: SINGLETONPROPERTY  
end tuple;  

proc instantiateLocalFrame(frame: LOCALFRAME, env: ENVIRONMENT): LOCALFRAME  
  instantiatedFrame: LOCALFRAME [] new LOCALFRAME[localBindings: {}, properties: SINGLETONPROPERTY {},  
                                        propertyTranslations: PROPERTYTRANSLATION {},  
                                        from: instantiateProperty(m, instantiatedFrame @ env)]  
  m: PROPERTYTRANSLATION [] the one element mi [] propertyTranslations that satisfies mi.from = m;  
  return mi.to  
end proc;  

instantiatedFrame.localBindings [] {LOCALBINDING[content: translateProperty(b.content), other fields from b]  
                                        [] b [] frame.localBindings};  

return instantiatedFrame  
end proc;
proc instantiateParameterFrame(frame: PARAMETERFRAME, env: ENVIRONMENT, singularThis: OBJECTOPT):
PARAMETERFRAME

note frame.superconstructorCalled must be true if and only if frame.kind is not constructorFunction.

instantiateFrame: PARAMETERFRAME [] new PARAMETERFRAME(localBindings: {}, kind: frame.kind,
    handling: frame.handling, callsSuperconstructor: frame.callsSuperconstructor,
superconstructorCalled: frame.superconstructorCalled, this: singularThis, returnType: frame.returnType[]

note properties will contain the set of all SINGLETONPROPERTY records found in the frame.
properties: SINGLETONPROPERTY {} [] {b.content | b [] frame.localBindings};

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

for each p [] frame.parameters do properties [] properties [] {p.var} end for each;
rest: VARIABLEOPT [] frame.rest;
if rest = none then properties [] properties [] {rest} end if;
propertyTranslations: PROPERTYTRANSLATION {} [] {PROPERTYTRANSLATION[from: m,
    to: instantiateProperty(m, [instantiateFrame] @ env)] [] m [] properties};
proc translateProperty(m: SINGLETONPROPERTY): SINGLETONPROPERTY

mi: PROPERTYTRANSLATION [] the one element mi [] propertyTranslations that satisfies mi.from = m;
return mi.to
end proc;
instantiateFrame.localBindings [] {LOCALBINDING[content: translateProperty(b.content), other fields from b[]]
[] b [] frame.localBindings};
instantiateFrame.parameters [] [PARAMETER[var: translateProperty(op.var), default: op.default[]
    [] op [] frame.parameters];
if rest = none then instantiateFrame.rest [] none
else instantiateFrame.rest [] translateProperty(rest) end if;
return instantiateFrame
end proc;

10.16 Sealing

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

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

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

```plaintext
proc defaultArg(args: Object[], n: Integer, default: Object): Object
    if n < |args| then return default end if;
    arg: Object[] args[n];
    if arg = undefined then return default else return arg end if
end proc;
```

```plaintext
proc stdConstBinding(qname: QualifiedName, type: Class, value: Object): LocalBinding
    return LocalBinding[qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
        new Variable[type: type, value: value, immutable: true, setup: none, initializer: none[]]]
end proc;
```

```plaintext
proc stdExplicitConstBinding(qname: QualifiedName, type: Class, value: Object): LocalBinding
    return LocalBinding[qname: qname, accesses: readWrite, explicit: true, enumerable: false, content:
        new Variable[type: type, value: value, immutable: true, setup: none, initializer: none[]]]
end proc;
```

```plaintext
proc stdVarBinding(qname: QualifiedName, type: Class, value: Object): LocalBinding
    return LocalBinding[qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
        new Variable[type: type, value: value, immutable: false, setup: none, initializer: none[]]]
end proc;
```

```plaintext
proc stdFunction(qname: QualifiedName, call: Object[] SimpleInstance[] Object[] Phase[] Object,
    length: Integer): LocalBinding
    slots: Slot[] [] {new Slot[i] ivarFunctionLength, value: length[]};
    f: SimpleInstance[] new SimpleInstance[] localBindings: {}, archetype: FunctionPrototype, sealed: true,
    return LocalBinding[qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
        new Variable[type: Function, value: f, immutable: true, setup: none, initializer: none[]]]
end proc;
```

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

```plaintext
proc stdReserve(qname: QualifiedName, archetype: SimpleInstance): LocalBinding
    matchingBindings: LocalBinding[] [] {b | b archetype.localBindings such that b.qname = qname};
    return the one element of matchingBindings
end proc;
```

### 1211 Expressions

Some expression grammar productions in this chapter are parameterised (see section 5.14.4) by the grammar argument `[]`.  
`[]` {allowIn, noIn}

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

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

12.11.2 Identifiers

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

Syntax

Identifier []
  Identifier
  | get
  | set

Semantics

Name[Identifier]: STRING;
  Name[Identifier] : Identifier = Name[Identifier];
  Name[Identifier] : get = “get”;
  Name[Identifier] : set = “set”;

11.3 Qualified Identifiers

Syntax

SimpleQualifiedIdentifier []
  Identifier
  | Identifier :: Identifier
  | ReservedNamespace :: Identifier

ExpressionQualifiedIdentifier [] ParenExpression :: Identifier

QualifiedIdentifier []
  SimpleQualifiedIdentifier
  | ExpressionQualifiedIdentifier

Validation

OpenNamespaces[SimpleQualifiedIdentifier]: NAMESPACE{};

Strict[SimpleQualifiedIdentifier]: BOOLEAN;

proc Validate[SimpleQualifiedIdentifier] (ctx: CONTEXT, env: ENVIRONMENT)
  [SimpleQualifiedIdentifier [] Identifier] do
    OpenNamespaces[SimpleQualifiedIdentifier] . ctx.openNamespaces;
    Strict[SimpleQualifiedIdentifier] . ctx.strict;
SimpleQualifiedIdentifier :: Identifier

OpenNamespaces[SimpleQualifiedIdentifier] cxt.openNamespaces;

ReservedNamespace :: Identifier
do Evaluate Validate[ReservedNamespace](cxt, env) and ignore its result
end proc;

Strict[ExpressionQualifiedIdentifier] : BOOLEAN;

c proc Validate[ExpressionQualifiedIdentifier :: ParenExpression] (cxt: CONTEXT, env: ENVIRONMENT) Strict[ExpressionQualifiedIdentifier] cxt.strict;

Evaluate Validate[ParenExpression](cxt, env) and ignore its result
end proc;

Strict[QualifiedIdentifier] : BOOLEAN;

c proc Validate[QualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT) [QualifiedIdentifier] do

Strict[QualifiedIdentifier] cxt.strict;

Evaluate Validate[SimpleQualifiedIdentifier](cxt, env) and ignore its result;

Strict[QualifiedIdentifier] [ExpressionQualifiedIdentifier] do

Strict[QualifiedIdentifier] cxt.strict;

Evaluate Validate[ExpressionQualifiedIdentifier](cxt, env) and ignore its result
end proc;

(proc Evaluate[QualifiedIdentifier] (env: ENVIRONMENT, phase: PHASE): MULTINAME

[SimpleQualifiedIdentifier :: Identifier] do

return {ns:(Name[Identifier]) | ns ns OpenNamespaces[SimpleQualifiedIdentifier]};

[SimpleQualifiedIdentifier :: Identifier :: Identifier] do

multiname MULTINAME {ns:(Name[Identifier]) | ns ns OpenNamespaces[SimpleQualifiedIdentifier]};
a: OBJECT lexicalRead(env, multiline, phase);
if a NAMESPACE then

throw a TypeError exception — the qualifier must be a namespace
end if;

return {a:(Name[Identifier])};

)
do
  q: NAMESPACE [] Eval[ReservedNamespace](env, phase);
  return {q::(Name[Identifier])}
end proc;

proc Eval[ExpressionQualifiedIdentifier [] ParenExpression : : Identifier]
  (env: ENVIRONMENT, phase: PHASE): MULTINAME
  q: OBJECT [] readReference(Eval[ParenExpression](env, phase), phase);
  if q [] NAMESPACE then throw a TypeError exception — the qualifier must be a namespace
  end if;
  return {q::(Name[Identifier])}
end proc;

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

11.4 Primary Expressions

Syntax

PrimaryExpression []
  null
  | true
  | false
  | Number
  | String
  | this
  | RegularExpression
  | ReservedNamespace
  | ParenListExpression
  | ArrayLiteral
  | ObjectLiteral
  | FunctionExpression

ReservedNamespace []
  public
  | private

ParenExpression [] (AssignmentExpression allowin )

ParenListExpression []
  ParenExpression
  | ( ListExpression allowin, AssignmentExpression allowin )
Validation

**proc Validate[PrimaryExpression]** (ext: CONTEXT, env: ENVIRONMENT)

[PrimaryExpression [] null] do nothing;

[PrimaryExpression [] true] do nothing;

[PrimaryExpression [] false] do nothing;

[PrimaryExpression [] Number] do nothing;

[PrimaryExpression [] String] do nothing;

[PrimaryExpression [] this] do

  frame: PARAMETERFRAMEOPT [] getEnclosingParameterFrame(env);

  if frame = none then
    if ext.strict then
      throw a SyntaxError exception — this can be used outside a function only in non-strict mode
    end if
  end if

  elsif frame.kind = plainFunction then
    throw a SyntaxError exception — this function does not define this
  end if

[PrimaryExpression [] RegularExpression] do nothing;

[PrimaryExpression [] ReservedNamespace] do

  Evaluate Validate[ReservedNamespace](ext, env) and ignore its result;

[PrimaryExpression [] ParenListExpression] do

  Evaluate Validate[ParenListExpression](ext, env) and ignore its result;

[PrimaryExpression [] ArrayLiteral] do

  Evaluate Validate[ArrayLiteral](ext, env) and ignore its result;

[PrimaryExpression [] ObjectLiteral] do

  Evaluate Validate[ObjectLiteral](ext, env) and ignore its result;

[PrimaryExpression [] FunctionExpression] do

  Evaluate Validate[FunctionExpression](ext, env) and ignore its result

end proc;

**proc Validate[ReservedNamespace]** (ext: CONTEXT, env: ENVIRONMENT)

[ReservedNamespace [] public] do nothing;

[ReservedNamespace [] private] do

  if getEnclosingClass(env) = none then
    throw a SyntaxError exception — private is meaningful only inside a class
  end if

end proc;

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

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

Setup

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

**proc Setup[ReservedNamespace]** ()

  [ReservedNamespace [] public] do nothing;

  [ReservedNamespace [] private] do nothing

end proc;
Setup\[\text{ParenExpression}\] () propagates the call to Setup to nonterminals in the expansion of ParenExpression.

Setup\[\text{ParenListExpression}\] () propagates the call to Setup to nonterminals in the expansion of ParenListExpression.

Evaluation

\textbf{proc Eval}[\text{PrimaryExpression}] (env: \textsc{Environment}, phase: \textsc{Phase}): \textsc{ObjOrRef}

\begin{verbatim}
[PrimaryExpression] null] do return null;
[PrimaryExpression] true] do return true;
[PrimaryExpression] false] do return false;
[PrimaryExpression] Number] do return Value[Number];
[PrimaryExpression] String] do return Value[String];
[PrimaryExpression] this] do
  \begin{verbatim}
  frame: \textsc{ParameterFrameOpt} [] getEnclosingParameterFrame(env);
  if frame = none then return getPackageFrame(env) end if;
  note Validate ensured that frame.kind \neq \textsc{plainFunction} at this point.
  this: \textsc{ObjectOpt} [] frame.this;
  if this = none then
    note If Validate passed, this can be uninitialised only when phase = \textsc{compile}.
    throw a ConstantError exception — a constant expression cannot read an uninitialised this parameter
  end if;
  if not frame.superconstructorCalled then
    throw an UninitializedError exception — can’t access this from within a constructor before the
    superconstructor has been called
  end if;
  return this;
  [PrimaryExpression] RegularExpression] do
    return Body[RegularExpression] \oplus "#" \oplus Flags[RegularExpression];
  [PrimaryExpression] ReservedNamespace] do
    return Eval[ReservedNamespace](env, phase);
  [PrimaryExpression] ParenListExpression] do
    return Eval[ParenListExpression](env, phase);
  [PrimaryExpression] ArrayLiteral] do return Eval[ArrayLiteral](env, phase);
  [PrimaryExpression] ObjectLiteral] do return Eval[ObjectLiteral](env, phase);
  [PrimaryExpression] FunctionExpression] do
    return Eval[FunctionExpression](env, phase);
end proc;
\end{verbatim}

\textbf{proc Eval}[\text{ReservedNamespace}] (env: \textsc{Environment}, phase: \textsc{Phase}): \textsc{Namespace}

\begin{verbatim}
[ReservedNamespace] public] do return public;
[ReservedNamespace] private] do
  c: \textsc{ClassOpt} [] getEnclosingClass(env);
  note Validate already ensured that c \neq none.
  return c.privateNamespace
end proc;
\end{verbatim}

Eval\[\text{ParenExpression}\] (env: \textsc{Environment}, phase: \textsc{Phase}): \textsc{ObjOrRef} propagates the call to Eval to nonterminals in the expansion of ParenExpression.
11.5 Function Expressions

Syntax

\[ \text{FunctionExpression} \]
\begin{align*}
\text{function} & \quad \text{FunctionCommon} \\
\text{|} & \quad \text{function} \quad \text{Identifier} \quad \text{FunctionCommon}
\end{align*}

Validation

\[ F[\text{FunctionExpression}] : \text{UNINSTANTIATEDFUNCTION}; \]

\textbf{proc \ Validate[FunctionExpression]} (\text{ext}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT})
\[ \text{[FunctionExpression} \quad \text{function} \quad \text{FunctionCommon}] \text{ do} \]
\begin{align*}
\text{kind}: & \quad \text{STATICFUNCTION} \quad \text{kind} \quad \text{plainFunction}; \\
\text{if not} & \text{ext}\text{.strict} \text{ and} \text{Plain[FunctionCommon]} \text{ then} \text{kind} \quad \text{uncheckedFunction} \\
\text{end if}; \\
F[\text{FunctionExpression}] \quad \text{ValidateStaticFunction[FunctionCommon]}(\text{ext}, \text{env}, \text{kind});
\end{align*}

\[ \text{[FunctionExpression} \quad \text{function} \quad \text{Identifier} \quad \text{FunctionCommon}] \text{ do} \]
\begin{align*}
\text{v}: & \quad \text{VARIABLE} \quad \text{new} \quad \text{VARIABLE} \quad \text{type}: \quad \text{Function}, \quad \text{value}: \quad \text{none}, \quad \text{immutable}: \quad \text{true}, \quad \text{setup}: \quad \text{none}, \quad \text{initializer}: \quad \text{busy}[] \\
\text{b}: & \quad \text{LOCALBINDING} \quad \text{LOCALBINDING} \quad \text{name}: \quad \text{public}: (\text{Name[Identifier]}, \text{accesses}: \quad \text{readWrite}, \text{explicit}: \quad \text{false}, \\
& \text{enumerable}: \quad \text{true}, \quad \text{content}: \text{v}[]) \\
\text{compileFrame}: & \quad \text{LOCALFRAME} \quad \text{new} \quad \text{LOCALFRAME} \quad \text{localBindings}: \quad \{b\}[] \\
\text{kind}: & \quad \text{STATICFUNCTION} \quad \text{kind} \quad \text{plainFunction}; \\
\text{if not} & \text{ext}\text{.strict} \text{ and} \text{Plain[FunctionCommon]} \text{ then} \text{kind} \quad \text{uncheckedFunction} \\
\text{end if}; \\
F[\text{FunctionExpression}] \quad \text{ValidateStaticFunction[FunctionCommon]}(\text{ext}, \text{compileFrame} \oplus \text{env}, \text{kind})
\end{align*}

end proc;

Setup

\textbf{proc \ Setup[FunctionExpression]} ()
\[ \text{[FunctionExpression} \quad \text{function} \quad \text{FunctionCommon}] \text{ do} \]
Evaluate \text{Setup[FunctionCommon]}() \text{ and ignore its result;}

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

Evaluation

  [FunctionExpression [] function FunctionCommon] do
    if phase = compile then
      throw a ConstantError exception — a function expression is not a constant expression because it can evaluate to different values
    end if;
    return instantiateFunction(F[FunctionExpression], env);
  [FunctionExpression [] function Identifier FunctionCommon] do
    if phase = compile then
      throw a ConstantError exception — a function expression is not a constant expression because it can evaluate to different values
    end if;
  b: LOCALBINDING [] LOCALBINDING[name: public::(Name[Identifier]), accesses: readWrite, explicit: false, enumerable: true, content: v]
  runtimeFrame: LOCALFRAME [] new LOCALFRAME[localBindings: {b}]
  f: SIMPLEINSTANCE [] instantiateFunction(F[FunctionExpression], [runtimeFrame] ⊕ env);
  v.value [] f;
  return / f
end proc;

11.6 Object Literals

Syntax

ObjectLiteral [] { FieldList }

FieldList []
  «empty»
  | NonemptyFieldList

NonemptyFieldList []
  LiteralField
  | LiteralField , NonemptyFieldList

LiteralField [] FieldName : AssignmentExpression

FieldName []
  QualifiedIdentifier
  | String
  | Number
  | ParenExpression

Validation

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

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

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

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

Setup

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

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

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

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

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

Evaluation

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

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

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

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

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

Syntax

\[
\text{ArrayLiteral} \ [ \ 	ext{ElementList} ]
\]

\[
\text{ElementList} \ []
\]

- «empty»
- \(\text{LiteralElement}\)
- \(, \ 	ext{ElementList}\)
- \(\text{LiteralElement}, \ 	ext{ElementList}\)

\[
\text{LiteralElement} \ [] \ 	ext{AssignmentExpression} \ ^\text{llowin}
\]

Validation

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

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

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

Setup

\[
\text{Setup}[\text{ArrayLiteral}] () \text{ propagates the call to } \text{Setup} \text{ to nonterminals in the expansion of } \text{ArrayLiteral}.
\]

\[
\text{Setup}[\text{ElementList}] () \text{ propagates the call to } \text{Setup} \text{ to nonterminals in the expansion of } \text{ElementList}.
\]

\[
\text{Setup}[\text{LiteralElement}] () \text{ propagates the call to } \text{Setup} \text{ to nonterminals in the expansion of } \text{LiteralElement}.
\]

Evaluation

\[
\text{proc Eval}[\text{ArrayLiteral} [] [ \ 	ext{ElementList} ]] (\text{env}: \text{ENVIRONMENT}, \text{phase}: \text{PHASE}): \text{OBJOrREF}
\]

\[
\text{if } \text{phase} = \text{compile then}
\]

\[
\text{throw a } \text{ConstantError} \text{ exception — an array literal is not a constant expression because it evaluates to a new object each time it is evaluated}
\]

\[
\text{end if;}
\]

\[
\text{o: OBJECT [] construct(Array, [], phase)};
\]

\[
\text{length: INTEGER [] Eval[ElementList]}(\text{env}, 0, \text{phase});
\]

\[
\text{Evaluate writeArrayPrivateLength(o, length, phase) and ignore its result;}
\]

\[
\text{return o}
\]

\[
\text{end proc;}
\]

\[
\text{proc Eval[ElementList]} (\text{env}: \text{ENVIRONMENT}, \text{length: INTEGER}, \text{o: OBJECT, phase: {run}}): \text{INTEGER}
\]

\[
[\text{ElementList} [] «empty»] \text{ do return length;}
\]

\[
[\text{ElementList} [] \text{LiteralElement}] \text{ do}
\]

\[
\text{Evaluate Eval[LiteralElement]}(\text{env}, \text{length, o, phase}) \text{ and ignore its result;}
\]

\[
\text{return length + 1;}
\]

\[
[\text{ElementList}_0 [], \text{ElementList}_1] \text{ do}
\]

\[
\text{return Eval[ElementList}_1](\text{env}, \text{length + 1, o, phase});
\]
(ElementList₀, LiteralElement, ElementList₁) do
    Evaluate Eval[LiteralElement](env, length, o, phase) and ignore its result;
    return Eval[ElementList₁](env, length + 1, o, phase)
end proc;

proc Eval[LiteralElement, AssignmentExpressionallowin] (env: ENVIRONMENT, length: INTEGER, o: OBJECT, phase: {run})
    value: OBJECT = readReference(Eval[AssignmentExpressionallowin](env, phase), phase);
    Evaluate indexWrite(o, length, value, phase) and ignore its result
end proc;

11.8 Super Expressions

Syntax

SuperExpression []
    super
    | super ParenExpression

Validation

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

Setup

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

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

proc makeLimitedInstance(r: OBJORREF, c: CLASS, phase: PHASE): OPTIONAL LIMIT
  o: OBJECT [] readReference(r, phase);
  limit: CLASSOPT [] c.super;
    note Validate ensured that limit cannot be none at this point.
  coerced: OBJECT [] coerce(o, limit);
    if coerced = null then return null end if;
  return LIMITEDINSTANCE[Instance: coerced, limit: limit[]
end proc;

11.9 Postfix Expressions

Syntax

PostfixExpression []
  AttributeExpression
  | FullPostfixExpression
  | ShortNewExpression

AttributeExpression []
  SimpleQualifiedIdentifier
  | AttributeExpression PropertyOperator
  | AttributeExpression Arguments

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

FullNewExpression [] new FullNewSubexpression Arguments
FullNewSubexpression []
  PrimaryExpression
  | QualifiedIdentifier
  | FullNewExpression
  | FullNewSubexpression PropertyOperator
  | SuperExpression PropertyOperator

ShortNewExpression [] new ShortNewSubexpression

Validation

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

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

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

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

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

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

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

Setup

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

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

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

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

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

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

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

Evaluation

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

[AttributeExpression o SimpleQualifiedIdentifier] do
  m: MULTINAME [] Eval[SimpleQualifiedIdentifier](env, phase);
  return LEXICALREFERENCE [env, variableMultiname: m, strict: Strict][SimpleQualifiedIdentifier] []

[AttributeExpression o AttributeExpression, PropertyOperator] do
  a: OBJECT [] readReference(Eval[AttributeExpression])(env, phase), phase);
  return Eval[PropertyOperator](env, a, phase);

[AttributeExpression o AttributeExpression, Arguments] do
  r: OBJORREF [] Eval[AttributeExpression](env, phase);
  f: OBJECT [] readReference(r, phase);
  base: OBJECT;
  case r of
    OBJECT [] LEXICALREFERENCE do base [] null;
    DOTREFERENCE [] BracketReference do base [] r.base
  end case;
  args: OBJECT [] Eval[Arguments](env, phase);
  return call(base, f, args, phase)
end proc;

proc Eval[FullPostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF

[FullPostfixExpression o PrimaryExpression] do
  return Eval[PrimaryExpression](env, phase);

[FullPostfixExpression o ExpressionQualifiedIdentifier] do
  m: MULTINAME [] Eval[ExpressionQualifiedIdentifier](env, phase);
  return LEXICALREFERENCE [env, variableMultiname: m, strict: Strict][ExpressionQualifiedIdentifier] []

[FullPostfixExpression o FullNewExpression] do
  return Eval[FullNewExpression](env, phase);

[FullPostfixExpression o FullPostfixExpression, PropertyOperator] do
  a: OBJECT [] readReference(Eval[FullPostfixExpression])(env, phase), phase);
  return Eval[PropertyOperator](env, a, phase);

[FullPostfixExpression o SuperExpression PropertyOperator] do
  a: OBJOPTIONALLIMIT [] Eval[SuperExpression](env, phase);
  return Eval[PropertyOperator](env, a, phase);

[FullPostfixExpression o FullPostfixExpression, Arguments] do
  r: OBJORREF [] Eval[FullPostfixExpression](env, phase);
  f: OBJECT [] readReference(r, phase);
  base: OBJECT;
  case r of
    OBJECT [] LEXICALREFERENCE do base [] null;
    DOTREFERENCE [] BracketReference do base [] r.base
  end case;
  args: OBJECT [] Eval[Arguments](env, phase);
  return call(base, f, args, phase);
[FullPostfixExpression \(\) PostfixExpression \([\) no line break] \(+\) do
  if phase = compile then
    throw a ConstantError exception — ++ cannot be used in a constant expression
  end if;
  r: OBJORREF \(\) Eval[PostfixExpression](env, phase);
  a: OBJECT \(\) readReference(r, phase);
  b: OBJECT \(\) plus(a, phase);
  c: OBJECT \(\) add(b, \(\)'_''164\(\) phase);
  Evaluate writeReference(r, c, phase) and ignore its result;
  return b;
end proc;

[FullPostfixExpression \(\) PostfixExpression \([\) no line break] \(-\) do
  if phase = compile then
    throw a ConstantError exception — -- cannot be used in a constant expression
  end if;
  r: OBJORREF \(\) Eval[PostfixExpression](env, phase);
  a: OBJECT \(\) readReference(r, phase);
  b: OBJECT \(\) plus(a, phase);
  c: OBJECT \(\) subtract(b, \(\)'_''164\(\) phase);
  Evaluate writeReference(r, c, phase) and ignore its result;
  return b;
end proc;

  f: OBJECT \(\) readReference(Eval[FullNewSubexpression](env, phase), phase);
  args: OBJECT[] \(\) Eval[Arguments](env, phase);
  return construct(f, args, phase)
end proc;

  [FullNewSubexpression \(\) PrimaryExpression] do
    return Eval[PrimaryExpression](env, phase);
  [FullNewSubexpression \(\) QualifiedIdentifier] do
    m: MULTINAME \(\) Eval[QualifiedIdentifier](env, phase);
    return LEXICALREFERENCE[env: env, variableMultiname: m, strict: Strict[QualifiedIdentifier]]
  [FullNewSubexpression \(\) FullNewExpression] do
    return Eval[FullNewExpression](env, phase);
  [FullNewSubexpression, FullNewSubexpression, PropertyOperator] do
    a: OBJECT \(\) readReference(Eval[FullNewSubexpression], (env, phase), phase);
    return Eval[PropertyOperator](env, a, phase);
  [FullNewSubexpression, SuperExpression PropertyOperator] do
    a: OBJECT OPTIONALLIMIT \(\) Eval[SuperExpression](env, phase);
    return Eval[PropertyOperator](env, a, phase)
end proc;

  f: OBJECT \(\) readReference(Eval[ShortNewSubexpression](env, phase), phase);
  return construct(f, []), phase
end proc;

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

Syntax

\[
\text{PropertyOperator} [ \\
\quad . \quad \text{QualifiedIdentifier} \\
\quad | \quad \text{Brackets}
\]

\[
\text{Brackets} [ \\
\quad [ ] \\
\quad | \quad [ \text{ListExpression}\text{allowIn} ] \\
\quad | \quad [ \text{ExpressionsWithRest} ]
\]

\[
\text{Arguments} [ \\
\quad ( ) \\
\quad | \quad \text{ParenListExpression} \\
\quad | \quad ( \text{ExpressionsWithRest} )
\]

\[
\text{ExpressionsWithRest} [ \\
\quad \text{RestExpression} \\
\quad | \quad \text{ListExpression}\text{allowIn}, \text{RestExpression}
\]

\[
\text{RestExpression} [ \\
\quad . . . \text{AssignmentExpression}\text{allowIn}
\]

Validation

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

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

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

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

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

Setup

\[
\text{Setup}[\text{PropertyOperator}] () \text{ propagates the call to Setup to nonterminals in the expansion of PropertyOperator.}
\]

\[
\text{Setup}[\text{Brackets}] () \text{ propagates the call to Setup to nonterminals in the expansion of Brackets.}
\]

\[
\text{Setup}[\text{Arguments}] () \text{ propagates the call to Setup to nonterminals in the expansion of Arguments.}
\]

\[
\text{Setup}[\text{ExpressionsWithRest}] () \text{ propagates the call to Setup to nonterminals in the expansion of ExpressionsWithRest.}
\]

\[
\text{Setup}[\text{RestExpression}] () \text{ propagates the call to Setup to nonterminals in the expansion of RestExpression.}
\]
Evaluation

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

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

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

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

a: OBJECT[] readReference(Eval[AssignmentExpression][allowIn])(env, phase);
length: INTEGER[] readLength(a, phase);
i: INTEGER[] 0;
args: OBJECT[][] [];

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

11.11 Unary Operators

Syntax

UnaryExpression []
  PostfixExpression
  | delete PostfixExpression
  | void UnaryExpression
  | typeof UnaryExpression
  | ++ PostfixExpression
  | -- PostfixExpression
  | + UnaryExpression
  | - UnaryExpression
  | ~ UnaryExpression
  | ! UnaryExpression

Validation

Strict[UnaryExpression]: BOOLEAN;

proc Validate[UnaryExpression](ext: CONTEXT, env: ENVIRONMENT)
[UnaryExpression [] PostfixExpression] do
  Evaluate Validate[PostfixExpression](ext, env) and ignore its result;
[UnaryExpression [] delete PostfixExpression] do
  Evaluate Validate[PostfixExpression](ext, env) and ignore its result;
Strict[UnaryExpression] [] ext.strict;
[UnaryExpression0 [] void UnaryExpression,] do
  Evaluate Validate[UnaryExpression](ext, env) and ignore its result;
[UnaryExpression0 [] typeof UnaryExpression,] do
  Evaluate Validate[UnaryExpression](ext, env) and ignore its result;
[UnaryExpression [] ++ PostfixExpression] do
  Evaluate Validate[PostfixExpression](ext, env) and ignore its result;
UnaryExpression -- PostfixExpression] do
Evaluate Validate[PostfixExpression](ctx, env) and ignore its result;
UnaryExpression  + UnaryExpression1] do
Evaluate Validate[UnaryExpression1](ctx, env) and ignore its result;
UnaryExpression  - UnaryExpression1] do
Evaluate Validate[UnaryExpression1](ctx, env) and ignore its result;
UnaryExpression  - NegatedMinLong do nothing;
UnaryExpression  ~ UnaryExpression1] do
Evaluate Validate[UnaryExpression1](ctx, env) and ignore its result;
UnaryExpression  ! UnaryExpression1] do
Evaluate Validate[UnaryExpression1](ctx, env) and ignore its result
end proc;

Setup

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

Evaluation

proc Eval[UnaryExpression](env: ENVIRONMENT, phase: PHASE): OBJORREF
UnaryExpression  PostfixExpression] do return Eval[PostfixExpression](env, phase);
UnaryExpression  delete PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — delete cannot be used in a constant expression
  end if;
  r: OBJORREF  Eval[PostfixExpression](env, phase);
  return deleteReference(r, Strict[UnaryExpression], phase);
UnaryExpression  void UnaryExpression1] do
  Evaluate readReference(Eval[UnaryExpression1](env, phase), phase) and ignore its result;
  return undefined;
UnaryExpression  typeof UnaryExpression1] do
  a: OBJECT  readReference(Eval[UnaryExpression1](env, phase), phase);
  c: CLASS  objectType(a);
  return c.typeOfString;
UnaryExpression  ++ PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — ++ cannot be used in a constant expression
  end if;
  r: OBJORREF  Eval[PostfixExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  b: OBJECT  plus(a, phase);
  c: OBJECT  add(b, 1qd, phase);
  Evaluate writeReference(r, c, phase) and ignore its result;
  return c;
[UnaryExpression] -- PostfixExpression] do
   if phase = compile then
      throw a ConstantError exception — -- cannot be used in a constant expression
   end if;
   r: OBJORRef [] Eval[PostfixExpression](env, phase);
   a: OBJECT [] readReference(r, phase);
   b: OBJECT [] plus(a, phase);
   c: OBJECT [] subtract(b, 1); Eval[PostfixExpression](env, phase);
   Evaluate writeReference(r, c, phase) and ignore its result;
   return c;
[UnaryExpression] + UnaryExpression1] do
   a: OBJECT [] readReference(Eval[UnaryExpression1](env, phase), phase);
   return plus(a, phase);
[UnaryExpression] - UnaryExpression1] do
   a: OBJECT [] readReference(Eval[UnaryExpression1](env, phase), phase);
   return minus(a, phase);
[UnaryExpression] - NegatedMinLong] do return (-2^63)long;
[UnaryExpression] ~ UnaryExpression1] do
   a: OBJECT [] readReference(Eval[UnaryExpression1](env, phase), phase);
   return bitNot(a, phase);
[UnaryExpression] ! UnaryExpression1] do
   a: OBJECT [] readReference(Eval[UnaryExpression1](env, phase), phase);
   return logicalNot(a, phase)
end proc;

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

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

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

proc minus(a: OBJECT, phase: PHASE): OBJECT
   x: GENERALNUMBER [] objectToGeneralNumber(a, phase);
   return generalNumberNegate(x)
end proc;

proc generalNumberNegate(x: GENERALNUMBER): GENERALNUMBER
   case x of
      LONG do return integerToLong(-x.value);
      ULONG do return integerToULong(-x.value);
      FLOAT32 do return float32Negate(x);
      FLOAT64 do return float64Negate(x)
   end case
end proc;
**11.12 Multiplicative Operators**

**Syntax**

```
MultiplicativeExpression []
  UnaryExpression
| MultiplicativeExpression * UnaryExpression
| MultiplicativeExpression / UnaryExpression
| MultiplicativeExpression % UnaryExpression
```

**Validation**

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

**Setup**

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

**Evaluation**

```
[MultiplicativeExpression [] UnaryExpression] do
  return Eval[UnaryExpression](env, phase);
[MultiplicativeExpression1 * MultiplicativeExpression2] do
  a: OBJECT [] readReference(Eval[MultiplicativeExpression1](env, phase), phase);
  b: OBJECT [] readReference(Eval[UnaryExpression2](env, phase), phase);
  return multiply(a, b, phase);
[MultiplicativeExpression1 / MultiplicativeExpression2] do
  a: OBJECT [] readReference(Eval[MultiplicativeExpression1](env, phase), phase);
  b: OBJECT [] readReference(Eval[UnaryExpression2](env, phase), phase);
  return divide(a, b, phase);
```
[MultiplicativeExpression_0 □ MultiplicativeExpression_1 % UnaryExpression] do
  a: OBJECT □ readReference(Eval[MultiplicativeExpression_0](env, phase));
  b: OBJECT □ readReference(Eval[UnaryExpression](env, phase));
  return remainder(a, b, phase)
end proc;

proc multiply(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL_NUMBER □ objectToGeneralNumber(a, phase);
y: GENERAL_NUMBER □ objectToGeneralNumber(b, phase);
if x □ LONG or y □ LONG then
  i: INTEGEROPT □ checkInteger(x);
  j: INTEGEROPT □ checkInteger(y);
  if i ≠ none and j ≠ none then
    k: INTEGER □ i/j;
    if x □ ULONG or y □ ULONG then return integerToULong(k)
  else return integerToLong(k)
  end if
end if
return float64Multiply(toFloat64(x), toFloat64(y))
end proc;

proc divide(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL_NUMBER □ objectToGeneralNumber(a, phase);
y: GENERAL_NUMBER □ objectToGeneralNumber(b, phase);
if x □ LONG or y □ LONG then
  i: INTEGEROPT □ checkInteger(x);
  j: INTEGEROPT □ checkInteger(y);
  if i ≠ none and j ≠ none and j ≠ 0 then
    q: RATIONAL □ i/j;
    if x □ ULONG or y □ ULONG then return rationalToULong(q)
  else return rationalToLong(q)
  end if
end if
return float64Divide(toFloat64(x), toFloat64(y))
end proc;

proc remainder(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL_NUMBER □ objectToGeneralNumber(a, phase);
y: GENERAL_NUMBER □ objectToGeneralNumber(b, phase);
if x □ LONG or y □ LONG then
  i: INTEGEROPT □ checkInteger(x);
  j: INTEGEROPT □ checkInteger(y);
  if i ≠ none and j ≠ none and j ≠ 0 then
    q: RATIONAL □ i/j;
    k: INTEGER □ q ≥ 0 ? kq: kq
    r: INTEGER □ i – jk;
    if x □ ULONG or y □ ULONG then return integerToULong(r)
  else return integerToLong(r)
  end if
end if
return float64Remainder(toFloat64(x), toFloat64(y))
end proc;
11.13 Additive Operators

Syntax

\[
\text{AdditiveExpression} = \begin{array}{l}
\text{MultiplicativeExpression} \\
\text{AdditiveExpression} + \text{MultiplicativeExpression} \\
\text{AdditiveExpression} - \text{MultiplicativeExpression}
\end{array}
\]

Validation

\[
\text{Validate}[\text{AdditiveExpression}] (\text{ctx: CONTEXT}, \text{env: ENVIRONMENT}) \text{ propagates the call to Validate to nonterminals in the expansion of AdditiveExpression.}
\]

Setup

\[
\text{Setup}[\text{AdditiveExpression}] () \text{ propagates the call to Setup to nonterminals in the expansion of AdditiveExpression.}
\]

Evaluation

\[
\begin{array}{l}
\text{proc Eval}[\text{AdditiveExpression}] (\text{env: ENVIRONMENT}, \text{phase: PHASE}): \text{OBJORRef} \\
\quad \text{[AdditiveExpression} = \text{MultiplicativeExpression}] \text{ do} \\
\quad \text{return Eval}[\text{MultiplicativeExpression}][\text{env}, \text{phase}]; \\
\quad \text{[AdditiveExpression} = \text{AdditiveExpression} + \text{MultiplicativeExpression}] \text{ do} \\
\quad \quad \text{a: OBJECT} \quad \text{readReference(Eval[AdditiveExpression]}][\text{env}, \text{phase}]); \\
\quad \quad \text{b: OBJECT} \quad \text{readReference(Eval[AdditiveExpression}][\text{env}, \text{phase}]); \\
\quad \quad \text{return add(a, b, phase);} \\
\quad \text{[AdditiveExpression} = \text{AdditiveExpression} - \text{MultiplicativeExpression}] \text{ do} \\
\quad \quad \text{a: OBJECT} \quad \text{readReference(Eval[AdditiveExpression]}][\text{env}, \text{phase}]); \\
\quad \quad \text{b: OBJECT} \quad \text{readReference(Eval[AdditiveExpression}][\text{env}, \text{phase}]); \\
\quad \quad \text{return subtract(a, b, phase);} \\
\end{array}
\]

\[
\begin{array}{l}
\text{proc add(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT} \\
\quad \text{ap: PRIMITIVEOBJECT} \quad \text{objectToPrimitive(a, none, phase);} \\
\quad \text{bp: PRIMITIVEOBJECT} \quad \text{objectToPrimitive(b, none, phase);} \\
\quad \text{if ap \ CHAR16 \ String or bp \ CHAR16 \ String then} \\
\quad \quad \text{return objectToString(ap, phase) \& \& objectToString(bp, phase)} \\
\quad \end{array}
\]

\[
\begin{array}{l}
\quad x: \text{GENERALNUMBER} \quad \text{objectToGeneralNumber(ap, phase);} \\
\quad y: \text{GENERALNUMBER} \quad \text{objectToGeneralNumber(bp, phase);} \\
\quad \text{if x \ LONG \ ULONG or y \ LONG \ ULONG then} \\
\quad \quad i: \text{INTEGEROPT} \quad \text{checkInteger(x);} \\
\quad \quad j: \text{INTEGEROPT} \quad \text{checkInteger(y);} \\
\quad \quad \text{if i \# none and j \# none then} \\
\quad \quad \quad k: \text{INTEGER} \quad i + j; \\
\quad \quad \quad \text{if x \ ULONG or y \ ULONG then return integerToULong(k)} \\
\quad \quad \quad \text{else return integerToLong(k)} \\
\quad \quad \end{array}
\]

\[
\begin{array}{l}
\quad \text{end if} \\
\quad \text{end if} \\
\quad \text{return float64Add(toFloat64(x), toFloat64(y))} \\
\end{array}
\]

end proc;
proc subtract(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERAL NUMBER = objectToGeneralNumber(a, phase);
  y: GENERAL NUMBER = objectToGeneralNumber(b, phase);
  if x \in LONG \cup LONG or y \in LONG \cup LONG then
    i: INTEGER OPT = checkInteger(x);
    j: INTEGER OPT = checkInteger(y);
    if i \neq none and j \neq none then
      k: INTEGER = i - j;
      if x \in LONG \cup LONG or y \in LONG \cup LONG then return integerToULong(k)
      else return integerToLong(k)
    end if
  end if
  return float64Subtract(toFloat64(x), toFloat64(y))
end proc;

11.14 Bitwise Shift Operators

Syntax

\[
\text{ShiftExpression} \equiv \text{AdditiveExpression} \\
| \text{ShiftExpression} \ll AdditiveExpression \\
| \text{ShiftExpression} \gg AdditiveExpression \\
| \text{ShiftExpression} \ggg AdditiveExpression
\]

Validation

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

Setup

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

Evaluation

proc Eval[ShiftExpression] (env: ENVIRONMENT, phase: PHASE):OBJOrRef
  [ShiftExpression \equiv AdditiveExpression] do
    return Eval[AdditiveExpression](env, phase);
  [ShiftExpression \ll ShiftExpression, \ll AdditiveExpression] do
    a: OBJECT \equiv readReference(Eval[ShiftExpression],(env, phase), phase);
    b: OBJECT \equiv readReference(Eval[AdditiveExpression],(env, phase), phase);
    return shiftLeft(a, b, phase);
  [ShiftExpression \gg ShiftExpression, \gg AdditiveExpression] do
    a: OBJECT \equiv readReference(Eval[ShiftExpression],(env, phase), phase);
    b: OBJECT \equiv readReference(Eval[AdditiveExpression],(env, phase), phase);
    return shiftRight(a, b, phase);
  [ShiftExpression \ggg ShiftExpression, \ggg AdditiveExpression] do
    a: OBJECT \equiv readReference(Eval[ShiftExpression],(env, phase), phase);
    b: OBJECT \equiv readReference(Eval[AdditiveExpression],(env, phase), phase);
    return shiftRightUnsigned(a, b, phase)
end proc;
proc shiftLeft(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT

x: GENERALNUMBER " objectToGeneralNumber(a, phase);
count: INTEGER " truncateToInteger(objectToGeneralNumber(b, phase));
case x of
  FLOAT32 □ FLOAT64 do
    count □ bitwiseAnd(count, 0x1F);
    i: {−2^31 ... 2^31 − 1} □ signedWrap32(bitwiseShift(truncateToInteger(x), count));
    return i64;
  LONG do
    count □ bitwiseAnd(count, 0x3F);
    i: {−2^63 ... 2^63 − 1} □ signedWrap64(bitwiseShift(x.value, count));
    return i_long;
  ULONG do
    count □ bitwiseAnd(count, 0x3F);
    i: {0 ... 2^64 − 1} □ unsignedWrap64(bitwiseShift(x.value, count));
    return i_ulong;
end case
end proc;

proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT

x: GENERALNUMBER " objectToGeneralNumber(a, phase);
count: INTEGER " truncateToInteger(objectToGeneralNumber(b, phase));
case x of
  FLOAT32 □ FLOAT64 do
    i: {−2^31 ... 2^31 − 1} □ signedWrap32(truncateToInteger(x));
    count □ bitwiseAnd(count, 0x1F);
    i □ bitwiseShift(i, −count);
    return i64;
  LONG do
    count □ bitwiseAnd(count, 0x3F);
    i: {−2^63 ... 2^63 − 1} □ bitwiseShift(x.value, −count);
    return i_long;
  ULONG do
    count □ bitwiseAnd(count, 0x3F);
    i: {−2^63 ... 2^63 − 1} □ bitwiseShift(signedWrap64(x.value), −count);
    return (unsignedWrap64(i))ulong;
end case
end proc;
proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL.NUMBER // objectToGeneralNumber(a, phase);
count: INTEGER // truncateToInteger(objectToGeneralNumber(b, phase));
case x of
  FLOAT32 // FLOAT64 do
    i: {0 ... 2^32 – 1} // unsignedWrap32(truncateToInteger(x));
    count // bitwiseAnd(count, 0x1F);
    i // bitwiseShift(i, –count);
    return i64;
  LONG do
    count // bitwiseAnd(count, 0x3F);
    i: {0 ... 2^64 – 1} // bitwiseShift(unsignedWrap64(x.value), –count);
    return (signedWrap64(i))_long;
  ULONG do
    count // bitwiseAnd(count, 0x3F);
    i: {0 ... 2^64 – 1} // bitwiseShift(x.value, –count);
    return i_ulong
end case
end proc;

11.15 Relational Operators

Syntax

RelationalExpression allowIn ShiftExpression
  | RelationalExpression allowIn < ShiftExpression
  | RelationalExpression allowIn > ShiftExpression
  | RelationalExpression allowIn <= ShiftExpression
  | RelationalExpression allowIn >= ShiftExpression
  | RelationalExpression allowIn is ShiftExpression
  | RelationalExpression allowIn as ShiftExpression
  | RelationalExpression allowIn in ShiftExpression
  | RelationalExpression allowIn instanceof ShiftExpression

RelationalExpression noIn ShiftExpression
  | RelationalExpression noIn < ShiftExpression
  | RelationalExpression noIn > ShiftExpression
  | RelationalExpression noIn <= ShiftExpression
  | RelationalExpression noIn >= ShiftExpression
  | RelationalExpression noIn is ShiftExpression
  | RelationalExpression noIn as ShiftExpression
  | RelationalExpression noIn instanceof ShiftExpression

Validation

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

Setup

Setup[RelationalExpression] () propagates the call to Setup to nonterminals in the expansion of RelationalExpression.
Evaluation

proc Eval[RelationalExpression{p}](env: ENVIRONMENT, phase: PHASE): OBJORREF

[RelationalExpression{p} \ ShiftExpression] do
    return Eval[ShiftExpression](env, phase);

[RelationalExpression{p} \ RelationalExpression{q}, < ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    return isLess(a, b, phase);

[RelationalExpression{p} \ RelationalExpression{q}, > ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    return isLess(b, a, phase);

[RelationalExpression{p} \ RelationalExpression{q}, <= ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    return isLessOrEqual(a, b, phase);

[RelationalExpression{p} \ RelationalExpression{q}, >= ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    return isLessOrEqual(b, a, phase);

[RelationalExpression{p} \ RelationalExpression{q}, is ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    c: CLASS \ objectToClass(b);
    return is(a, c);

[RelationalExpression{p} \ RelationalExpression{q}, as ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    c: CLASS \ objectToClass(b);
    return coerceOrNull(a, b, phase);

[RelationalExpression{p} \ RelationalExpression{q}, allowing in ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    return hasProperty(b, a, false, phase);

[RelationalExpression{p} \ RelationalExpression{q}, instanceof ShiftExpression] do
    a: OBJECT \ readReference(Eval[RelationalExpression{p}](env, phase), phase);
    b: OBJECT \ readReference(Eval[ShiftExpression](env, phase), phase);
    if b \ CLASS then return is(a, b)
    elsif is(b, PrototypeFunction) then
        prototype: OBJECT \ dotRead(b, {public::"prototype"}, phase);
        return prototype \ archetypes(a)
    else throw a TypeError exception
    end if
end proc;
proc isLess(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
   ap: PRIMITIVEOBJECT  objectToPrimitive(a, hintNumber, phase);
   bp: PRIMITIVEOBJECT  objectToPrimitive(b, hintNumber, phase);
   if ap $\text{CHAR16} \text{ STRING}$ and bp $\text{CHAR16} \text{ STRING}$ then
     return toString(ap) < toString(bp)
   end if;
   return generalNumberCompare(objectToGeneralNumber(ap, phase), objectToGeneralNumber(bp, phase)) = less
end proc;

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

11.16 Equality Operators

Syntax

\[
\text{EqualityExpression}^0 \ 
| \ 
\text{RelationalExpression}^1
\]

\[
\text{EqualityExpression}^0 \ = \ RelationalExpression^1
\]

\[
\text{EqualityExpression}^0 \ != \ RelationalExpression^1
\]

\[
\text{EqualityExpression}^0 \ == \ RelationalExpression^1
\]

\[
\text{EqualityExpression}^0 \ != \ RelationalExpression^1
\]

Validation

Validate[EqualityExpression^0] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of EqualityExpression^0.

Setup

Setup[EqualityExpression^0] () propagates the call to Setup to nonterminals in the expansion of EqualityExpression^0.

Evaluation

proc Eval[EqualityExpression^0] (env: ENVIRONMENT, phase: PHASE): OBJORREF
   [EqualityExpression^0] | RelationalExpression^1
   return Eval[RelationalExpression^1](env, phase);

[EqualityExpression^0 | EqualityExpression^1 | RelationalExpression^0]
   a: OBJECT \ readReference(Eval[EqualityExpression^0](env, phase), phase);
   b: OBJECT \ readReference(Eval[RelationalExpression^0](env, phase), phase);
   return isEqual(a, b, phase);

[EqualityExpression^0 | EqualityExpression^1, != RelationalExpression^0]
   a: OBJECT \ readReference(Eval[EqualityExpression^0](env, phase), phase);
   b: OBJECT \ readReference(Eval[RelationalExpression^0](env, phase), phase);
   return not isEqual(a, b, phase);
[EqualityExpression\,\] \[EqualityExpression\] \[==\] RelationalExpression do
  \ a: OBJECT \ readReference(Eval[EqualityExpression])(env, phase), phase;\n  \ b: OBJECT \ readReference(Eval[RelationalExpression])(env, phase), phase;\n  \ return isStrictlyEqual(a, b, phase);\n[EqualityExpression\,\] EqualityExpression \[!=\] RelationalExpression do
  \ a: OBJECT \ readReference(Eval[EqualityExpression])(env, phase), phase;\n  \ b: OBJECT \ readReference(Eval[RelationalExpression])(env, phase), phase;\n  \ return not isStrictlyEqual(a, b, phase)
end proc;

proc isEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  \ case a of
    \ Undefined \ \ Null do return b \ Undefined \ \ Null; \n  \ Boolean do
    if b \ Boolean then return a = b
    else return isEqual(objectToGeneralNumber(a, phase), b, phase)
  \ end if;
  \ GeneralNumber do
    \ bp: PrimitiveObject \ objectToPrimitive(b, none, phase);
  \ case bp of
    \ Undefined \ \ Null do return false;
    \ Boolean \ \ GeneralNumber \ \ CharSet \ \ String do
      \ return generalNumberCompare(a, objectToGeneralNumber(bp, phase)) = equal
    \ end case;
  \ CharSet \ \ String do
    \ bp: PrimitiveObject \ objectToPrimitive(b, none, phase);
  \ case bp of
    \ Undefined \ \ Null do return false;
    \ Boolean \ \ GeneralNumber do
      \ return generalNumberCompare(objectToGeneralNumber(a, phase),
        objectToGeneralNumber(bp, phase)) = equal;
  \ CharSet \ \ String do
    \ return toString(a) = toString(bp)
  \ end case;
  \ Namespace \ \ CompoundAttribute \ \ Class \ \ MethodClosure \ \ SimpleInstance \ \ Date \ \ RegExp \ \ Package do
    \ case b of
      \ Undefined \ \ Null do return false;
      \ Namespace \ \ CompoundAttribute \ \ Class \ \ MethodClosure \ \ SimpleInstance \ \ Date \ \ RegExp \ \ Package do
        \ return isStrictlyEqual(a, b, phase);
      \ Boolean \ \ GeneralNumber \ \ CharSet \ \ String do
        \ ap: PrimitiveObject \ objectToPrimitive(a, none, phase);
        \ return isEqual(ap, b, phase)
    \ end case
  \ end case
end proc;

proc isStrictlyEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  \ if a \ GeneralNumber and b \ GeneralNumber then
    \ return generalNumberCompare(a, b) = equal
  \ else return a = b
  \ end if
end proc;
11.17 Binary Bitwise Operators

Syntax

\[
\text{BitwiseAndExpression} \Rightarrow \begin{cases} \\
\text{EqualityExpression} \\
\text{BitwiseAndExpression} \& \text{EqualityExpression} \\
\end{cases}
\]

\[
\text{BitwiseXorExpression} \Rightarrow \begin{cases} \\
\text{BitwiseXorExpression} \lor \text{BitwiseAndExpression} \\
\text{BitwiseXorExpression} \& \text{BitwiseXorExpression} \\
\end{cases}
\]

\[
\text{BitwiseOrExpression} \Rightarrow \begin{cases} \\
\text{BitwiseOrExpression} \lor \text{BitwiseXorExpression} \\
\text{BitwiseOrExpression} \lor \text{BitwiseOrExpression} \\
\end{cases}
\]

Validation

\text{Validate} [\text{BitwiseAndExpression}] (\text{ctx: CONTEXT}, \text{env: ENVIRONMENT}) propagates the call to \text{Validate} to nonterminals in the expansion of \text{BitwiseAndExpression}.

\text{Validate} [\text{BitwiseXorExpression}] (\text{ctx: CONTEXT}, \text{env: ENVIRONMENT}) propagates the call to \text{Validate} to nonterminals in the expansion of \text{BitwiseXorExpression}.

\text{Validate} [\text{BitwiseOrExpression}] (\text{ctx: CONTEXT}, \text{env: ENVIRONMENT}) propagates the call to \text{Validate} to nonterminals in the expansion of \text{BitwiseOrExpression}.

Setup

\text{Setup} [\text{BitwiseAndExpression}] () propagates the call to \text{Setup} to nonterminals in the expansion of \text{BitwiseAndExpression}.

\text{Setup} [\text{BitwiseXorExpression}] () propagates the call to \text{Setup} to nonterminals in the expansion of \text{BitwiseXorExpression}.

\text{Setup} [\text{BitwiseOrExpression}] () propagates the call to \text{Setup} to nonterminals in the expansion of \text{BitwiseOrExpression}.

Evaluation

\text{proc} \text{Eval} [\text{BitwiseAndExpression}] (\text{env: ENVIRONMENT}, \text{phase: PHASE}: \text{OBJORREF} [\text{BitwiseAndExpression} \& \text{EqualityExpression}] do
  \text{return} \text{Eval} [\text{EqualityExpression}](\text{env}, \text{phase});
end proc;

\text{proc} \text{Eval} [\text{BitwiseXorExpression}] (\text{env: ENVIRONMENT}, \text{phase: PHASE}: \text{OBJORREF} [\text{BitwiseXorExpression} \lor \text{BitwiseAndExpression}] do
  \text{return} \text{Eval} [\text{BitwiseAndExpression}](\text{env}, \text{phase});
end proc;
proc Eval(BitwiseOrExpression) (env: ENVIRONMENT, phase: PHASE): OBJORREF
[BitwiseOrExpression] do
  return Eval(BitwiseOrExpression)(env, phase);
[BitwiseOrExpression] | BitwiseXorExpression] do
  a: OBJECT[] readReference(Eval(BitwiseXorExpression)(env, phase), phase);
  b: OBJECT[] readReference(Eval(BitwiseXorExpression)(env, phase), phase);
  return bitOr(a, b, phase)
end proc;

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER objectToGeneralNumber(a, phase);
y: GENERALNUMBER objectToGeneralNumber(b, phase);
if x[] LONG or y[] LONG then
  i: {−2^63 ... 2^63 − 1} signedWrap64(truncateToInteger(x));
  j: {−2^63 ... 2^63 − 1} signedWrap64(truncateToInteger(y));
  k: {−2^63 ... 2^63 − 1} bitwiseAnd(i, j);
  if x[] ULONG or y[] ULONG then return (unsignedWrap64(k))ulong
  else return klong
end if
else
  i: {−2^31 ... 2^31 − 1} signedWrap32(truncateToInteger(x));
  j: {−2^31 ... 2^31 − 1} signedWrap32(truncateToInteger(y));
  return (bitwiseAnd(i, j))f64
end if
end proc;

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER objectToGeneralNumber(a, phase);
y: GENERALNUMBER objectToGeneralNumber(b, phase);
if x[] LONG or y[] LONG then
  i: {−2^63 ... 2^63 − 1} signedWrap64(truncateToInteger(x));
  j: {−2^63 ... 2^63 − 1} signedWrap64(truncateToInteger(y));
  k: {−2^63 ... 2^63 − 1} bitwiseXor(i, j);
  if x[] ULONG or y[] ULONG then return (unsignedWrap64(k))ulong
  else return klong
end if
else
  i: {−2^31 ... 2^31 − 1} signedWrap32(truncateToInteger(x));
  j: {−2^31 ... 2^31 − 1} signedWrap32(truncateToInteger(y));
  return (bitwiseXor(i, j))f64
end if
end proc;
proc bitOr(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER ¶ objectToGeneralNumber(a, phase);
y: GENERALNUMBER ¶ objectToGeneralNumber(b, phase);
if x LONG or y LONG then
  i: {−2^{63} ... 2^{63} − 1} ¶ signedWrap64(truncateToInteger(x));
j: {−2^{63} ... 2^{63} − 1} ¶ signedWrap64(truncateToInteger(y));
k: {−2^{63} ... 2^{63} − 1} ¶ bitwiseOr(i, j);
end if
else return \( k \) LONG
end if
else
  i: {−2^{31} ... 2^{31} − 1} ¶ signedWrap32(truncateToInteger(x));
j: {−2^{31} ... 2^{31} − 1} ¶ signedWrap32(truncateToInteger(y));
return (bitwiseOr(i, j)) ULONG
end if
end proc;

11.18 Binary Logical Operators

Syntax

LogicalAndExpression\( ^{0} \) ¶
  BitwiseOrExpression\( ^{1} \)
  | LogicalAndExpression\( ^{0} \) \&\& BitwiseOrExpression\( ^{1} \)

LogicalXorExpression\( ^{0} \) ¶
  LogicalAndExpression\( ^{0} \)
  | LogicalXorExpression\( ^{0} \) ^^ LogicalAndExpression\( ^{0} \)

LogicalOrExpression\( ^{0} \) ¶
  LogicalXorExpression\( ^{0} \)
  | LogicalOrExpression\( ^{0} \) || LogicalXorExpression\( ^{0} \)

Validation

Validate[LogicalAndExpression\( ^{0} \)] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of LogicalAndExpression\( ^{0} \).

Validate[LogicalXorExpression\( ^{0} \)] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of LogicalXorExpression\( ^{0} \).

Validate[LogicalOrExpression\( ^{0} \)] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of LogicalOrExpression\( ^{0} \).

Setup

Setup[LogicalAndExpression\( ^{0} \)] () propagates the call to Setup to nonterminals in the expansion of LogicalAndExpression\( ^{0} \).

Setup[LogicalXorExpression\( ^{0} \)] () propagates the call to Setup to nonterminals in the expansion of LogicalXorExpression\( ^{0} \).

Setup[LogicalOrExpression\( ^{0} \)] () propagates the call to Setup to nonterminals in the expansion of LogicalOrExpression\( ^{0} \).
Evaluation

proc Eval[LogicalAndExpression\[2\]] (\textit{env}: ENVIRONMENT, \textit{phase}: PHASE): OBJORREF
    [LogicalAndExpression\[1\] \& BitwiseOrExpression\[1\]] do
        return Eval[BitwiseOrExpression\[1\]](\textit{env}, \textit{phase});
    [LogicalAndExpression\[0\] \& LogicalAndExpression\[1\] \& BitwiseOrExpression\[1\]] do
        a: OBJECT \& readReference(Eval[LogicalAndExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase});
        if objectToBoolean(a) then
            return readReference(Eval[BitwiseOrExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase})
        else return a
    end if
end proc;

proc Eval[LogicalXorExpression\[2\]] (\textit{env}: ENVIRONMENT, \textit{phase}: PHASE): OBJORREF
    [LogicalXorExpression\[1\] \& LogicalAndExpression\[1\] \& LogicalXorExpression\[1\]] do
        return Eval[LogicalAndExpression\[1\]](\textit{env}, \textit{phase});
    [LogicalXorExpression\[0\] \& LogicalXorExpression\[1\] \& LogicalAndExpression\[1\] \& LogicalXorExpression\[1\]] do
        a: OBJECT \& readReference(Eval[LogicalXorExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase});
        b: OBJECT \& readReference(Eval[LogicalAndExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase});
        ba: BOOLEAN \& objectToBoolean(a);
        bb: BOOLEAN \& objectToBoolean(b);
        return ba xor bb
end proc;

proc Eval[LogicalOrExpression\[2\]] (\textit{env}: ENVIRONMENT, \textit{phase}: PHASE): OBJORREF
    [LogicalOrExpression\[1\] \& LogicalXorExpression\[1\]] do
        return Eval[LogicalXorExpression\[1\]](\textit{env}, \textit{phase});
    [LogicalOrExpression\[0\] \& LogicalOrExpression\[1\] \& LogicalXorExpression\[1\]] do
        a: OBJECT \& readReference(Eval[LogicalOrExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase});
        if objectToBoolean(a) then return a
        else return readReference(Eval[LogicalXorExpression\[1\]](\textit{env}, \textit{phase}), \textit{phase})
    end if
end proc;

11.19 Conditional Operator

Syntax

\begin{verbatim}
ConditionalExpression\[2\] =
    LogicalOrExpression\[2\] | LogicalOrExpression\[1\] ? AssignmentExpression\[1\] : AssignmentExpression\[1\]
| NonAssignmentExpression\[1\]
NonAssignmentExpression\[1\] =
    LogicalOrExpression\[0\] | LogicalOrExpression\[1\] ? NonAssignmentExpression\[0\] : NonAssignmentExpression\[0\]
\end{verbatim}

Validation

Validate[ConditionalExpression\[2\]] (\textit{ext}: CONTEXT, \textit{env}: ENVIRONMENT) propagates the call to \texttt{Validate} to nonterminals in the expansion of \texttt{ConditionalExpression\[2\]}.

Validate[NonAssignmentExpression\[1\]] (\textit{ext}: CONTEXT, \textit{env}: ENVIRONMENT) propagates the call to \texttt{Validate} to nonterminals in the expansion of \texttt{NonAssignmentExpression\[1\]}.
Setup

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

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

Evaluation

proc Eval[ConditionalExpression\[\]] (env: ENVIRONMENT, phase: PHASE): OBJORREF

[ConditionalExpression\[\] | LogicalOrExpression\[\]) do
  return Eval[LogicalOrExpression\[\]](env, phase);

[ConditionalExpression\[\] ? AssignmentExpression\[\] | AssignmentExpression\[\]] do
  a: OBJECT | readReference(Eval[LogicalOrExpression\[\]](env, phase), phase);
  if objectToBoolean(a) then
    return readReference(Eval[AssignmentExpression\[\]](env, phase), phase)
  else return readReference(Eval[AssignmentExpression\[\]2](env, phase), phase)
end if
end proc;

proc Eval[NonAssignmentExpression\[\]] (env: ENVIRONMENT, phase: PHASE): OBJORREF

[NonAssignmentExpression\[\] | LogicalOrExpression\[\]) do
  return Eval[LogicalOrExpression\[\]](env, phase);

[NonAssignmentExpression\[\] ? NonAssignmentExpression\[\] | NonAssignmentExpression\[\]2] do
  a: OBJECT | readReference(Eval[LogicalOrExpression\[\]](env, phase), phase);
  if objectToBoolean(a) then
    return readReference(Eval[NonAssignmentExpression\[\]](env, phase), phase)
  else return readReference(Eval[NonAssignmentExpression\[\]2](env, phase), phase)
end if
end proc;

11.20 Assignment Operators

Syntax

AssignmentExpression\[\] | ConditionalExpression\[\]
| PostfixExpression = AssignmentExpression\[\]
| PostfixExpression CompoundAssignment AssignmentExpression\[\]
| PostfixExpression LogicalAssignment AssignmentExpression\[\]

CompoundAssignment\[\]

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

&=&
| ^^=
| ||=

Semantics

tag andEq;
tag xorEq;
tag orEq;

Validation

proc Validate[AssignmentExpression\()\] (ext: CONTEXT, env: ENVIRONMENT)
    do
        Evaluate Validate[ConditionalExpression\()\](ext, env) and ignore its result;
    [AssignmentExpression\()\] do
        Evaluate Evaluate[PostfixExpression](ext, env) and ignore its result;
        Evaluate Validate[AssignmentExpression\()\](ext, env) and ignore its result;
    PostfixExpression CompoundAssignment AssignmentExpression\()\] do
        Evaluate Evaluate[PostfixExpression](ext, env) and ignore its result;
        Evaluate Validate[AssignmentExpression\()\](ext, env) and ignore its result;
    PostfixExpression LogicalAssignment AssignmentExpression\()\] do
        Evaluate Evaluate[PostfixExpression](ext, env) and ignore its result;
        Evaluate Validate[AssignmentExpression\()\](ext, env) and ignore its result;
end proc;

Setup

proc Setup[AssignmentExpression\()\] ()
    do
        Evaluate Setup[ConditionalExpression\()\]() and ignore its result;
    [AssignmentExpression\()\] do
        Evaluate Setup[PostfixExpression](ext, env) and ignore its result;
        Evaluate Setup[AssignmentExpression\()\]() and ignore its result;
    PostfixExpression CompoundAssignment AssignmentExpression\()\] do
        Evaluate Setup[PostfixExpression]() and ignore its result;
        Evaluate Setup[AssignmentExpression\()\]() and ignore its result;
    PostfixExpression LogicalAssignment AssignmentExpression\()\] do
        Evaluate Setup[PostfixExpression]() and ignore its result;
        Evaluate Setup[AssignmentExpression\()\]() and ignore its result;
end proc;

Evaluation

proc Eval[AssignmentExpression\()\] (env: ENVIRONMENT, phase: PHASE): ObjOrRef
    do
        return Eval[ConditionalExpression\()\](env, phase);
AssignmentExpression[b] = AssignmentExpression[b]
if phase = compile then
  throw a ConstantError exception — assignment cannot be used in a constant expression
end if;
ra: OBJREF[] Eval[PostfixExpression](env, phase);
b: OBJECT[] readReference(Eval[AssignmentExpression[b]](env, phase), phase);
Evaluate writeReference(ra, b, phase) and ignore its result;
return b;
AssignmentExpression[b] = AssignmentExpression[b]
do
if phase = compile then
  throw a ConstantError exception — assignment cannot be used in a constant expression
end if;
rLeft: OBJREF[] Eval[PostfixExpression](env, phase);
oLeft: OBJECT[] readReference(rLeft, phase);
oRight: OBJECT[] readReference(Eval[AssignmentExpression[b]](env, phase), phase);
result: OBJECT[] Op[CompoundAssignment](oLeft, oRight, phase);
Evaluate writeReference(rLeft, result, phase) and ignore its result;
return result;
AssignmentExpression[b] = AssignmentExpression[b]
do
if phase = compile then
  throw a ConstantError exception — assignment cannot be used in a constant expression
end if;
rLeft: OBJREF[] Eval[PostfixExpression](env, phase);
oLeft: OBJECT[] readReference(rLeft, phase);
bLeft: BOOLEAN[] objectToBoolean(oLeft);
result: OBJECT[] oLeft;
case Operator[LogicalAssignment] of
  {andEq} do
    if bLeft then
      result[] readReference(Eval[AssignmentExpression[b]](env, phase), phase)
    end if;
  {xorEq} do
    bRight: BOOLEAN[] objectToBoolean(readReference(Eval[AssignmentExpression[b]](env, phase), phase));
    result[] bLeft xor bRight;
  {orEq} do
    if not bLeft then
      result[] readReference(Eval[AssignmentExpression[b]](env, phase), phase)
    end if
end case;
Evaluate writeReference(rLeft, result, phase) and ignore its result;
return result
end proc;
Op[CompoundAssignment]: OBJECT [ OBJECT PHASE ] OBJECT;
Op[CompoundAssignment] [ *= ] multiply;
Op[CompoundAssignment] [ /= ] divide;
Op[CompoundAssignment] [ %= ] remainder;
Op[CompoundAssignment] [ += ] add;
Op[CompoundAssignment] [ -= ] subtract;
Op[CompoundAssignment] [ <<= ] shiftLeft;
Op[CompoundAssignment] [ >>= ] shiftRight;
Op[CompoundAssignment] [ >>>= ] shiftRightUnsigned;
Op[CompoundAssignment] [ &= ] bitAnd;
Op[CompoundAssignment] [ ^= ] bitXor;
Op[CompoundAssignment] [ |= ] bitOr;

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

11.21 Comma Expressions

Syntax

ListExpression[ ]
  AssignmentExpression[ ]
| ListExpression[ ] , AssignmentExpression[ ]

Validation

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

Setup

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

Evaluation

proc Eval[ListExpression[ ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ListExpression[ ] AssignmentExpression[ ]] do
    return Eval[AssignmentExpression[ ]](env, phase);
  [ListExpression[ ] ListExpression[ ], AssignmentExpression[ ]] do
    Evaluate readReference(Eval[ListExpression[ ])(env, phase), phase) and ignore its result;
    return readReference(Eval[AssignmentExpression[ ]](env, phase), phase)
end proc;

proc EvalAsList[ListExpression[ ]] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
  [ListExpression[ ] AssignmentExpression[ ]] do
    elt: OBJECT [] readReference(Eval[AssignmentExpression[ ]](env, phase), phase);
    return [elt];
11.22 Type Expressions

Syntax

\[
\text{TypeExpression} \rightarrow \text{NonAssignmentExpression} \\
\]

Validation

\[
\text{Validate} [\text{TypeExpression}] (\text{ctx: CONTEXT}, \text{env: ENVIRONMENT}) \text{ propagates the call to Validate to nonterminals in the expansion of TypeExpression.}
\]

Setup and Evaluation

\[
\text{proc SetupAndEval} [\text{TypeExpression}] (\text{env: ENVIRONMENT}: \text{CLASS}) \\
\text{Evaluate Setup[NonAssignmentExpression]()} and ignore its result; \\
o: \text{OBJECT} \rightarrow \text{readReference(Eval[NonAssignmentExpression](env, compile), compile)}; \\
\text{return objectToClass}(o)
\]

end proc;

12 Statements

Syntax

\[
\text{Statement} \rightarrow \{ \text{abbrev, noShortIf, full} \} \\
\text{Statement} \rightarrow \text{ExpressionStatement Semicolon} \text{ Block} \text{ LabeledStatement} \text{ IfStatement} \text{ SwitchStatement} \text{ DoStatement} \text{ WhileStatement} \text{ ForStatement} \text{ WithStatement} \text{ ContinueStatement} \text{ BreakStatement} \text{ ReturnStatement} \text{ ThrowStatement} \text{ TryStatement} \\
\text{Substatement} \rightarrow \text{EmptyStatement} \text{ Statement} \text{ SimpleVariableDefinition Semicolon} \text{ Attributes [no line break] \{ Substatements \} }
\]
Substatements  
  «empty»  
  | SubstatementsPrefix Substatementabbrv

SubstatementsPrefix  
  «empty»  
  | SubstatementsPrefix Substatementfull

Semicolonabbrv  
  ;  
  | VirtualSemicolon  
  | «empty»

SemicolonnoShortIf  
  ;  
  | VirtualSemicolon  
  | «empty»

Semicolonfull  
  ;  
  | VirtualSemicolon

Validation

[Statement]  
  ExpressionStatement Semicolon  
  do  
    Evaluate Validate[ExpressionStatement](cxt, env) and ignore its result;

[Statement]  
  SuperStatement Semicolon  
  do  
    Evaluate Validate[SuperStatement](cxt, env) and ignore its result;

[Statement]  
  Block  
  do  
    Evaluate Validate[Block](cxt, env, jt, preinst) and ignore its result;

[Statement]  
  LabeledStatement  
  do  
    Evaluate Validate[LabeledStatement](cxt, env, sl, jt) and ignore its result;

[Statement]  
  IfStatement  
  do  
    Evaluate Validate[IfStatement](cxt, env, jt) and ignore its result;

[Statement]  
  SwitchStatement  
  do  
    Evaluate Validate[SwitchStatement](cxt, env, jt) and ignore its result;

[Statement]  
  DoStatement Semicolon  
  do  
    Evaluate Validate[DoStatement](cxt, env, sl, jt) and ignore its result;

[Statement]  
  WhileStatement  
  do  
    Evaluate Validate[WhileStatement](cxt, env, sl, jt) and ignore its result;

[Statement]  
  ForStatement  
  do  
    Evaluate Validate[ForStatement](cxt, env, sl, jt) and ignore its result;

[Statement]  
  WithStatement  
  do  
    Evaluate Validate[WithStatement](cxt, env, jt) and ignore its result;

[Statement]  
  ContinueStatement Semicolon  
  do  
    Evaluate Validate[ContinueStatement](jt) and ignore its result;

[Statement]  
  BreakStatement Semicolon  
  do  
    Evaluate Validate[BreakStatement](jt) and ignore its result;

[Statement]  
  ReturnStatement Semicolon  
  do  
    Evaluate Validate[ReturnStatement](cxt, env) and ignore its result;
[Statement\(^1\)] \[ \textit{ThrowStatement Semicolon}\(^1\)] do
Evaluate \textbf{Validate}[\textit{ThrowStatement}]\((\textit{ctx}, \textit{env})\) and ignore its result;

[Statement\(^1\)] \[ \textit{TryStatement} \] do
Evaluate \textbf{Validate}[\textit{TryStatement}]\((\textit{ctx}, \textit{env}, \textit{jt})\) and ignore its result
end proc;

\textbf{Enabled}[\textit{Substatement}\(^1\)] : \texttt{BOOLEAN};

c\textbf{proc} \textbf{Validate}[\textit{Substatement}\(^1\)] \((\textit{ctx}: \text{\textit{CONTEXT}}, \textit{env}: \text{\textit{ENVIRONMENT}}, \textit{sl}: \text{\texttt{LABEL}}\{\}, \textit{jt}: \text{\texttt{JUMP\_TARGETS}})\)

[Substatement\(^1\)] \[ \textit{EmptyStatement} \] do \textbf{nothing};

[Substatement\(^1\)] \[ \textit{Statement} \] do
Evaluate \textbf{Validate}[\textit{Statement}]\((\textit{ctx}, \textit{env}, \textit{sl}, \textit{jt}, \texttt{false})\) and ignore its result;

[Substatement\(^1\)] \[ \textit{SimpleVariableDefinition Semicolon}\(^1\)] do
Evaluate \textbf{Validate}[\textit{SimpleVariableDefinition}]\((\textit{ctx}, \textit{env})\) and ignore its result;

[Substatement\(^1\)] \[ \textit{Attributes} \] \{\} \{ \textit{Substatements} \} \[ \texttt{false} \] do
Evaluate \textbf{Validate}[\textit{Attributes}]\((\textit{ctx}, \textit{env})\) and ignore its result;

Evaluate \textbf{Setup}[\textit{Attributes}]() and ignore its result;

\textit{attr}: \texttt{ATTRIBUTE} \[ \textbf{Eval}[\textit{Attributes}]\((\textit{env}, \text{\texttt{compile}})\);

\textbf{if} \textit{attr} \[ \texttt{BOOLEAN} \] \textbf{then}
\hspace{1em} \textbf{throw} \texttt{TypeError} \textit{exception} — attributes other than \texttt{true} and \texttt{false} may be used in a statement but not a substatement
\textbf{end if};

\textbf{Enabled}[\textit{Substatement}\(^1\)] \[ \texttt{false} \] \textbf{if} \textit{attr} \textbf{then}
Evaluate \textbf{Validate}[\textit{Substatements}]\((\textit{ctx}, \textit{env}, \textit{jt})\) and ignore its result
\textbf{end if}
\textbf{end proc};

c\textbf{proc} \textbf{Validate}[\textit{Substatements}] \((\textit{ctx}: \text{\textit{CONTEXT}}, \textit{env}: \text{\textit{ENVIRONMENT}}, \textit{jt}: \text{\texttt{JUMP\_TARGETS}})\)

[\textit{Substatements}] \[ \texttt{empty} \] do \textbf{nothing};

[\textit{Substatements}] \[ \texttt{SubstatementsPrefix Substatement}\(^{\texttt{abbrev}}\)] do
Evaluate \textbf{Validate}[\textit{SubstatementsPrefix}]\((\textit{ctx}, \textit{env}, \textit{jt})\) and ignore its result;

Evaluate \textbf{Validate}[\textit{Substatement}\(^{\texttt{abbrev}}\)]\((\textit{ctx}, \textit{env}, \{\}, \textit{jt})\) and ignore its result
\textbf{end proc};

c\textbf{proc} \textbf{Validate}[\textit{SubstatementsPrefix}] \((\textit{ctx}: \text{\textit{CONTEXT}}, \textit{env}: \text{\textit{ENVIRONMENT}}, \textit{jt}: \text{\texttt{JUMP\_TARGETS}})\)

[\textit{SubstatementsPrefix}] \[ \texttt{empty} \] do \textbf{nothing};

[\textit{SubstatementsPrefix}] \[ \texttt{SubstatementsPrefix Substatement}\(^{\texttt{full}}\)] do
Evaluate \textbf{Validate}[\textit{SubstatementsPrefix}]\((\textit{ctx}, \textit{env}, \textit{jt})\) and ignore its result;

Evaluate \textbf{Validate}[\textit{Substatement}\(^{\texttt{full}}\)]\((\textit{ctx}, \textit{env}, \{\}, \textit{jt})\) and ignore its result
\textbf{end proc};

\textbf{Setup}

\textbf{Setup}[\textit{Statement}\(^1\)] \() \text{ propagates the call to } \textbf{Setup}\text{ to nonterminals in the expansion of } \textit{Statement}\(^1\).
[Substatement[ Attributes [no line break] { Substatements } ]] do
if Enabled[Substatement[ ] then
  Evaluate Setup[Substatements]() and ignore its result
end if
end proc;

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

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

proc Setup[Semicolon[ ]] ()
  [Semicolon[ ] : ] do nothing;
  [Semicolon[ ] VirtualSemicolon] do nothing;
  [Semicolon[ ] abbrev «empty»] do nothing;
  [Semicolon[ ] noShortIf «empty»] do nothing
end proc;

Evaluation

proc Eval[Statement[ ] (env: ENVIRONMENT, d: OBJECT): OBJECT
    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);
    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);
end proc;

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

proc Eval[SubstatementsPrefix](env: ENVIRONMENT, d: OBJECT): OBJECT
    [SubstatementsPrefix □ «empty»] do return d;
    [SubstatementsPrefix □ SubstatementsPrefix| Substatement]| alt do
        o: OBJECT □ Eval[SubstatementsPrefix|](env, d);
        return Eval[Substatement]| alt(env, o)
    end proc;

12.1 Empty Statement

Syntax

EmptyStatement □ ;

12.2 Expression Statement

Syntax

ExpressionStatement □ [lookahead □ function, {}] ListExpression| allowIn

Validation

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

Setup

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

Evaluation

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

12.3 Super Statement

Syntax

SuperStatement □ super Arguments
Validation

\[\text{proc Validate}[\text{SuperStatement} \to \text{super} \text{ Arguments}] (\text{ext}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT})\]

\[\text{frame: PARAMETERFRAMEOPT} \to \text{getEnclosingParameterFrame}(\text{env});\]

\[\text{if} \ \text{frame} = \text{none} \text{ or} \ \text{frame.kind} \neq \text{constructorFunction} \text{ then}\]

\[\text{throw a SyntaxError exception — a super statement is meaningful only inside a constructor}\]

\[\text{end if;}\]

\[\text{Evaluate Validate[Arguments]}(\text{ext}, \text{env}) \text{ and ignore its result};\]

\[\text{frame.callsSuperconstructor} \to \text{true}\]

\end proc;

Setup

\[\text{Setup[SuperStatement]()} \text{ propagates the call to Setup to nonterminals in the expansion of SuperStatement.}\]

Evaluation

\[\text{proc Eval}[\text{SuperStatement} \to \text{super} \text{ Arguments}] (\text{env}: \text{ENVIRONMENT}): \text{OBJECT}\]

\[\text{frame: PARAMETERFRAMEOPT} \to \text{getEnclosingParameterFrame}(\text{env});\]

\[\text{note} \ \text{Validate already ensured that frame \neq \text{none} \text{ and} \ frame.kind = \text{constructorFunction}.}\]

\[\text{args: OBJECT[]} \to \text{Eval[Arguments]}(\text{env}, \text{run});\]

\[\text{if} \ \text{frame.superconstructorCalled} = \text{true} \text{ then}\]

\[\text{throw a ReferenceError exception — the superconstructor cannot be called twice}\]

\[\text{end if;}\]

\[c: \text{CLASS} \to \text{getEnclosingClass}(\text{env});\]

\[\text{this: OBJECTOPT} \to \text{frame.this};\]

\[\text{note} \ \text{this \ SIMPEINSTANCE};\]

\[\text{Evaluate callInit(this, c.super, args, run)} \text{ and ignore its result};\]

\[\text{frame.superconstructorCalled} \to \text{true};\]

\[\text{return this}\]

\end proc;

12.4 Block Statement

Syntax

\[\text{Block} \to \{\text{Directives}\}\]

Validation

\[\text{CompileFrame[Block]}: \text{LOCALFRAME};\]

\[\text{Preinstantiate[Block]}: \text{BOOLEAN};\]

\[\text{proc ValidateUsingFrame[Block} \to \{\text{Directives}\}\]

\[\text{(ext}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT}, \text{jt}: \text{Jumptargets}, \text{preinst}: \text{BOOLEAN}, \text{frame}: \text{FRAME})\]

\[\text{localCxt}: \text{CONTEXT} \to \text{new CONTEXT[strict: ext.strict, opennamespaces: ext.opennamespaces]}\]

\[\text{Evaluate Validate[Directives]}(\text{localCxt}, [\text{frame}] \odot \text{env}, \text{jt}, \text{preinst}, \text{none}) \text{ and ignore its result}\]

\end proc;
proc Validate[Block [] { Directives }] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN) {
    compileFrame: LOCALFRAME = new LOCALFRAME
    localBindings: {}[]
    CompileFrame[Block][] compileFrame;
    Preinstantiate[Block][] preinst;
    Evaluate ValidateUsingFrame[Block](cxt, env, jt, preinst, compileFrame) and ignore its result
end proc;

Setup

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

Evaluation

proc Eval[Block [] { Directives }] (env: ENVIRONMENT, d: OBJECT): OBJECT {
    compileFrame: LOCALFRAME = CompileFrame[Block];
    runtimeFrame: LOCALFRAME =
    if Preinstantiate[Block] then runtimeFrame = compileFrame
    else runtimeFrame = instantiateLocalFrame(compileFrame, env) end if;
    return Eval[Directives][runtimeFrame ⊕ env, d]
end proc;

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

12.5 Labeled Statements

Syntax

LabeledStatement® Identifier : Substatement®

Validation

proc Validate[LabeledStatement® Identifier : Substatement®] {
    name: STRING = Name[Identifier];
    if name ∉ jt.breakTargets then
        throw a SyntaxError exception — nesting labeled statements with the same label is not permitted
    end if;
    jt2: JUMPTARGETS = breakTargets: jt.breakTargets [] {name},
    continueTargets: jt.continueTargets[]
    Evaluate Validate[Substatement®](cxt, env, sl [] {name}, jt2) and ignore its result
end proc;

Setup

proc Setup[LabeledStatement® Identifier : Substatement®] () {
    Evaluate Setup[Substatement®]() and ignore its result
end proc;
Evaluation

proc Eval[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
  end if
end try
end proc;

12.6 If Statement

Syntax

IfStatement\[ if ParenListExpression Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\f SHORT if

Validation

proc Validate[IfStatement\[ if ParenListExpression Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\f SHORT if\]

Setup

Setup[IfStatement\[ if ParenListExpression Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\f SHORT if\]] () propagates the call to Setup to nonterminals in the expansion of IfStatement\[ if ParenListExpression Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\f SHORT if\].

Evaluation

proc Eval[IfStatement\[ if ParenListExpression Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\]\[ if ParenListExpression Substatement\f SHORT if else Substatement\f SHORT if\]] (env: ENVIRONMENT, d: OBJECT): OBJECT
if objectToBoolean(o) then return Eval[Substatement](env, d)
else return d
end if;
12.7 Switch Statement

Semantics

tuple SWITCHKEY
  key: OBJECT
end tuple;

SWITCHGUARD = SWITCHKEY {default} OBJECT;

Syntax

SwitchStatement □ switch ParenListExpression { CaseElements }

CaseElements □
  «empty»
  □ CaseLabel
  □ CaseLabel CaseElementsPrefix CaseElement

CaseElementsPrefix □
  «empty»
  □ CaseElementsPrefix CaseElement

CaseElement □
  Directive
  □ CaseLabel

CaseLabel □
  case ListExpression:
  □ default:

Validation

CompileFrame[SwitchStatement]□ LOCALFRAME;
proc Validate[SwitchStatement  []  switch ParenListExpression  { CaseElements  } ]
  (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS)

if NDefaults[CaseElements] > 1 then
  throw a SyntaxError exception — a case statement may have at most one default clause
end if;

Evaluate Validate[ParenListExpression](ext, env) and ignore its result;

jt2: JUMP_TARGETS [] JUMP_TARGETS[breakTargets: jt.breakTargets [] {default},
  continueTargets: jt.continueTargets[]]

compileFrame: LOCAL_FRAME [] new LOCAL_FRAME[localBindings: {}[]]
CompileFrame[SwitchStatement] [] compileFrame;
localCxt: CONTEXT [] new CONTEXT[strict: ext.strict, openNamespaces: ext.openNamespaces[]]
Evaluate Validate[CaseElements](localCxt, [compileFrame] ⊕ env, jt2) and ignore its result
end proc;

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

Validate[CaseElements] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS) propagates the call to Validate to
nonterminals in the expansion of CaseElements.

NDefaults[CaseElementsPrefix]: INTEGER;
  NDefaults[CaseElementsPrefix [] «empty»] = 0;
NDefaults[CaseElementsPrefix [] CaseLabel] = NDefaults[CaseLabel];
NDefaults[CaseElementsPrefix [] CaseLabel CaseElementsPrefix CaseElement\abbrev]
  = NDefaults[CaseElementsPrefix] + NDefaults[CaseElement\abbrev];

Validate[CaseElementsPrefix] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS) propagates the call to Validate to
nonterminals in the expansion of CaseElementsPrefix.

NDefaults[CaseElement\abbrev]: INTEGER;
  NDefaults[CaseElement\abbrev [] Directive\abbrev] = 0;
NDefaults[CaseElement\abbrev [] CaseLabel] = NDefaults[CaseLabel];

proc Validate[CaseElement\abbrev] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS)
  [CaseElement\abbrev [] Directive\abbrev] do
    Evaluate Validate[Directive\abbrev](ext, env, jt, false, none) and ignore its result;
  [CaseElement\abbrev [] CaseLabel] do
    Evaluate Validate[CaseLabel](ext, env, jt) and ignore its result
  end proc;

NDefaults[CaseLabel]: INTEGER;
  NDefaults[CaseLabel [] case ListExpression\allowin ] = 0;
NDefaults[CaseLabel [] default : ] = 1;

proc Validate[CaseLabel] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS)
  [CaseLabel [] case ListExpression\allowin ] do
    Evaluate Validate[ListExpression\allowin ](ext, env) and ignore its result;
  [CaseLabel [] default : ] do nothing
end proc;
Setup

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

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

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

Setup[CaseLabel^0] () propagates the call to Setup to nonterminals in the expansion of CaseLabel^0.

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

Evaluation

proc Eval[SwitchStatement] [] switch ParenListExpression { CaseElements } ]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
key: OBJECT [] readReference(Eval[ParenListExpression](env, run), run);
compileFrame: LOCALFRAME [] CompileFrame[SwitchStatement];
runtimeFrame: LOCALFRAME [] instantiateLocalFrame(compileFrame, env);
runtimeEnv: ENVIRONMENT [] runtimeFrame ⊕ env;
result: SWITCHGUARD [] Eval[CaseElements](runtimeEnv, SWITCHKEY[key: key][d]);
if result OBJECT then return result end if;
note result = SWITCHKEY[key: key]
result () Eval[CaseElements](runtimeEnv, default, d);
if result OBJECT then return result end if;
note result = default;
return d
end proc;

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

  [CaseElementsPrefix [] «empty»] do return guard;
  [CaseElementsPrefix[] CaseElementsPrefix[] CaseElement^abbrev] do
    guard2: SWITCHGUARD [] Eval[CaseElementsPrefix](env, guard, d);
    return Eval[CaseElement^abbrev](env, guard2, d)
end proc;

proc Eval[CaseLabel^0] (env: ENVIRONMENT, guard: SWITCHGUARD, d: OBJECT): SWITCHGUARD
  [CaseLabel^0 [] Directive^2] do
    case guard of
      SWITCHKEY [] {default} do return guard;
      OBJECT do return Eval[Directive^3](env, guard)
    end case;
    [CaseLabel do return Eval[CaseLabel](env, guard, d)
end proc;

```plaintext
[CaseLabel [] case ListExpression allow :] do
  case guard of
    {default} [] `OBJECT` do return guard;
    `SWITCHKEY` do
      label: `OBJECT` [] `readReference(Eval[ListExpression allow](env, run), run)`;
      if isStrictlyEqual(guard, key, label, run) then return d
      else return guard
    end if
  end case;

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

12.8 Do-While Statement

Syntax

```
DoStatement [] do Substatement `abbrev while` ParenListExpression
```

Validation

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

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

Setup

```
Setup[DoStatement] () propagates the call to Setup to nonterminals in the expansion of DoStatement.
```
Evaluation

\[
\text{proc } \text{Eval}\left[ \text{DoStatement } \rightarrow \text{ do } \text{Substatement}^\text{abbrev} \text{ while } \text{ParenListExpression} \right] \\text{(env: Environment, } d: \text{ Object): Object}
\]

\[
\text{try}
\]

\[
d1: \text{Object } \rightarrow d;
\]

\[
\text{while true do}
\]

\[
\text{try } d1 \rightarrow \text{Eval}\left[ \text{Substatement}^\text{abbrev}(\text{env}, d1) \right]
\]

\[
\text{catch } x: \text{SemanticException do}
\]

\[
\text{if } x \rightarrow \text{ CONTINUE and } x.\text{label} \rightarrow \text{Labels[DoStatement]} \text{ then } d1 \rightarrow x.\text{value}
\]

\[
\text{end if}
\]

\[
\text{end try;}
\]

\[
o: \text{Object } \rightarrow \text{readReference}(\text{Eval}\left[ \text{ParenListExpression}\right](\text{env, run}), \text{run});
\]

\[
\text{if not objectToBoolean(o) then return } d1 \text{ end if}
\]

\[
\text{end while}
\]

\[
\text{catch } x: \text{SemanticException do}
\]

\[
\text{if } x \rightarrow \text{BREAK and } x.\text{label} = \text{default then return } x.\text{value} \text{ else throw } x \text{ end if}
\]

\[
\text{end try}
\]

\[
\text{end proc;}
\]

12.9 While Statement

Syntax

\[
\text{WhileStatement}^\text{[i]} \rightarrow \text{while } \text{ParenListExpression } \text{Substatement}^\text{[i]}
\]

Validation

\[
\text{Labels[WhileStatement}^\text{[i]}]: \text{LABEL}\{};
\]

\[
\text{proc Validate}\left[ \text{WhileStatement}^\text{[i]} \rightarrow \text{while } \text{ParenListExpression } \text{Substatement}^\text{[i]} \right] \\text{(ext: CONTEXT, env: Environment, } sl: \text{ LABEL}\{}, \text{jt: JUMPTARGETS)}
\]

\[
\text{continueLabels: LABEL}\{} \rightarrow sl \rightarrow \text{default};
\]

\[
\text{Labels[WhileStatement}^\text{[i]}]: \text{LABEL}\{}; \text{continueLabels;}
\]

\[
\text{jt2: JUMPTARGETS$^2$ JUMPTARGETS$^2$ breakTargets: jt.breakTargets$^2$ \rightarrow \text{default},}
\]

\[
\text{continueTargets: jt.continueTargets} \rightarrow \text{continueLabels$^2$}
\]

\[
\text{Evaluate Validate[ParenListExpression]}(\text{ext, env}) \text{ and ignore its result;}
\]

\[
\text{Evaluate Validate[Substatement}^\text{[i]}](\text{ext, env, } \{}), \text{jt2}\text{) and ignore its result}
\]

\[
\text{end proc;}
\]

Setup

\[
\text{Setup[WhileStatement}^\text{[i]}]() \text{ propagates the call to Setup to nonterminals in the expansion of WhileStatement}^\text{[i]}.
\]
Evaluation

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

### 12.10 For Statements

**Syntax**

```
ForStatement =
  for (ForInitializer; OptionalExpression; OptionalExpression) Substatement
  | for (ForInBinding in ListExpression) Substatement
```

**ForInitializer**

```
«empty»
| ListExpression
| VariableDefinition
| Attributes [no line break] VariableDefinition
```

**ForInBinding**

```
PostfixExpression
| VariableDefinitionKind VariableBinding
| Attributes [no line break] VariableDefinitionKind VariableBinding
```

**OptionalExpression**

```
ListExpression
| «empty»
```

**Validation**

```
Labels[ForStatement]: LABEL{);
CompileLocalFrame[ForStatement]: LOCALFRAME;
```
    [ForStatement^6] [] for ( ForInitializer; OptionalExpression_1; OptionalExpression_2 ) Substatement^7] do
        continueLabels: LABEL{}. [] sl[] {default};
        Labels[ForStatement^5] [] continueLabels;
        jt2: JUMP TARGETS [] JUMP TARGETS[break Targets: jt.break Targets [] {default},
            continue Targets: jt.continue Targets [] continueLabels[]
            compileLocalFrame: LOCAL FRAME [] new LOCAL FRAME[local Bindings: {}][]
        CompileLocalFrame[ForStatement^6] [] compileLocalFrame;
        compile Env: ENVIRONMENT [] [compileLocalFrame] @ env;
        Evaluate Validate[ForInitializer](ext, compile Env) and ignore its result;
        Evaluate Validate[OptionalExpression_1](ext, compile Env) and ignore its result;
        Evaluate Validate[OptionalExpression_2](ext, compile Env) and ignore its result;
        Evaluate Validate[Substatement^5](ext, compile Env, {}, jt2) and ignore its result;
    [ForStatement^6] [] for ( ForIn Binding in ListExpression^allow) Substatement^5] do
        continueLabels: LABEL{}. [] sl[] {default};
        Labels[ForStatement^5] [] continueLabels;
        jt2: JUMP TARGETS [] JUMP TARGETS[break Targets: jt.break Targets [] {default},
            continue Targets: jt.continue Targets [] continueLabels[]
            CompileLocalFrame[ForStatement^6] [] compileLocalFrame;
            compile Env: ENVIRONMENT [] [compileLocalFrame] @ env;
            Evaluate Validate[ForIn Binding](ext, compile Env) and ignore its result;
            Evaluate Validate[Substatement^5](ext, compile Env, {}, jt2) and ignore its result
end proc;

Enabled[ForInitializer]: BOOLEAN;

proc Validate[ForInitializer] (ext: CONTEXT, env: ENVIRONMENT)
    [ForInitializer] [empty] do nothing;
    [ForInitializer] ListExpression^allow do
        Evaluate Validate[ListExpression^allow](ext, env) and ignore its result;
    [ForInitializer] VariableDefinition^allow do
        Evaluate Validate[VariableDefinition^allow](ext, env, none) and ignore its result;
        [ForInitializer] Attributes [no line break] VariableDefinition^allow do
            Evaluate Validate[Attributes](ext, env) and ignore its result;
            Evaluate setup[Attributes] and ignore its result;
            attr: ATTRIBUTE [] Eval[Attributes](env, compile);
            Enabled[ForInitializer] [] attr # false;
            if attr # false then
                Evaluate Validate[VariableDefinition^allow](ext, env, attr) and ignore its result
end if
end proc;

proc Validate[ForIn Binding] (ext: CONTEXT, env: ENVIRONMENT)
    [ForIn Binding] PostfixExpression] do
        Evaluate Validate[PostfixExpression](ext, env) and ignore its result;
    [ForIn Binding] VariableDefinitionKind VariableBinding^allow do
        Evaluate Validate[VariableBinding^allow](ext, env, none, Immutable[VariableDefinitionKind], true) and ignore its result;
ForInBinding Attributes [no line break] VariableDefinitionKind VariableBinding\(^{\text{noIn}}\) do
Evaluate Validate[Attributes](\text{cxt}, \text{env}) and ignore its result;
Evaluate Setup[Attributes]() and ignore its result;
\(\text{attr} : \text{ATTRIBUTE} \) \(\text{Eval}[\text{Attributes}](\text{env}, \text{compile})\);
if \(\text{attr} = \text{false}\) then
  throw an AttributeError exception — the \text{false} attribute cannot be applied to a for-in variable definition
end if;
Evaluate Validate[VariableBinding\(^{\text{noIn}}\](\text{cxt}, \text{env}, \text{attr}, \text{Immutable}[\text{VariableDefinitionKind}], \text{true}) and ignore its result
end proc;

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

Setup

Setup[ForStatement\(^{2}\)] () propagates the call to Setup to nonterminals in the expansion of ForStatement\(^{2}\).

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

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

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

\[
\text{proc } \text{Eval[ForStatement]}(\text{env: \text{ENVIRONMENT}, d: \text{OBJECT}}) : \text{OBJECT} \\
\text{[ForStatement]} \quad \text{for} \ (\text{ForInitializer} ; \text{OptionalExpression}_1 ; \text{OptionalExpression}_2) \ \text{Substatement} \quad \text{do} \\
\text{runtimeLocalFrame: \text{LOCAL FRAME} } \quad \text{instantiateLocalFrame(CompileLocalFrame[ForStatement], env);} \\
\text{runtimeEnv: \text{ENVIRONMENT} } \quad \text{[runtimeLocalFrame] } \oplus \text{ env;} \\
\text{try} \\
\text{Evaluate \text{Eval[ForInitializer]}(runtimeEnv) and ignore its result;} \\
\text{d1: \text{OBJECT} } \quad \text{d}; \\
\text{while objectToBoolean(readReference(\text{Eval[OptionalExpression]})(runtimeEnv, run), run)) do} \\
\text{try} \ d1 \quad \text{Eval[Substatement]}(\text{runtimeEnv}, d1) \\
\text{catch x: \text{SEMANTIC EXCEPTION} do} \\
\quad \text{if x } \text{CONTINUE} \ \text{and x.label } \in \text{Labels[ForStatement]} \quad \text{then} \\
\quad \quad \quad d1 \quad \text{x.value} \\
\quad \text{else throw x} \\
\quad \text{end if} \\
\text{end try;} \\
\text{Evaluate readReference(\text{Eval[OptionalExpression]})(runtimeEnv, run), run) and ignore its result} \\
\text{end while;} \\
\text{return d1} \\
\text{catch x: \text{SEMANTIC EXCEPTION} do} \\
\quad \text{if x } \text{BREAK} \ \text{and x.label } = \text{default then return x.value else throw x} \ \text{end if} \\
\text{end try;}
\]
[ForStatement]\[\text{for} \ (\text{ForInBinding in ListExpression}) \ \text{Substatement}] \text{do}
try
  \text{o: OBJECT} \ \text{readReference(Eval[ListExpression](env, run), run)};
  \text{c: CLASS} \ \text{objectType(o)};
  \text{oldIndices: OBJECT} \ \text{c.enumerate(o)};
  \text{remainingIndices: OBJECT} \ \text{oldIndices};
  \text{d1: OBJECT} \ \text{d};
  \text{while remainingIndices \text{\neq \{\}}} \text{do}
    \text{runtimeLocalFrame: LOCALFRAME} \ \text{instantiateLocalFrame(CompileLocalFrame(ForStatement), env)};
    \text{runtimeEnv: ENVIRONMENT} \ \text{[runtimeLocalFrame]} \oplus env;
    \text{index: OBJECT} \ \text{any element of remainingIndices};
    \text{remainingIndices} \ \text{remainingIndices} \ \text{\{index\}};
  \text{Evaluate WriteBinding(ForInBinding)(runtimeEnv, index) and ignore its result};
  \text{try d1} \ \text{Eval[Substatement](runtimeEnv, d1)}
  \text{catch x: SEMANTICEXCEPTION} \text{do}
    \text{if x \ CONTINUE and x.label \ Labels[ForStatement] then}
    \text{d1} \ \text{x.value}
    \text{else throw x}
  \text{end if}
  \text{end try};
  \text{newIndices: OBJECT} \ \text{c.enumerate(o)};
  \text{if newIndices \text{\neq oldIndices then}
    \text{The implementation may, at its discretion, add none, some, or all of the objects in the set difference
      newIndices \text{– oldIndices} to remainingIndices;}
    \text{The implementation may, at its discretion, remove none, some, or all of the objects in the set difference
      oldIndices \text{– newIndices} from remainingIndices;}
  \text{end if};
  \text{oldIndices} \ \text{newIndices}
\text{end while};
\text{return d1}
\text{catch x: SEMANTICEXCEPTION} \text{do}
  \text{if x \ BREAK and x.label = default then return x.value else throw x end if}
\text{end try}
\text{end proc};

\text{proc Eval[ForInitializer] (env: ENVIRONMENT)}
  \text{[ForInitializer} \ \text{\{empty\}} \text{do nothing};
  \text{[ForInitializer} \ \text{ListExpression} \text{do}
    \text{Evaluate readReference(Eval[ListExpression](env, run), run) and ignore its result};
  \text{[ForInitializer} \ \text{VariableDefinition} \text{do}
    \text{Evaluate Eval[VariableDefinition](env, undefined) and ignore its result};
  \text{[ForInitializer} \ \text{Attributes} \text{[no line break] VariableDefinition} \text{do}
    \text{if Enabled[ForInitializer] then}
      \text{Evaluate Eval[VariableDefinition](env, undefined) and ignore its result}
    \text{end if}
  \text{end if}
\text{end proc};

\text{proc WriteBinding[ForInBinding] (env: ENVIRONMENT, newValue: OBJECT)}
  \text{[ForInBinding} \ \text{PostfixExpression} \text{do}
    \text{r: OBJECT} \ \text{Eval[PostfixExpression](env, run)};
    \text{Evaluate writeReference(r, newValue, run) and ignore its result};
  \text{[ForInBinding} \ \text{VariableDefinitionKind VariableBinding} \text{do}
    \text{Evaluate WriteBinding[VariableBinding](env, newValue) and ignore its result};
\text{end proc};
12.11 With Statement

Syntax

WithStatement with ParenListExpression Substatement

Validation

CompileLocalFrame[WithStatement]: LOCALFRAME;

Validate[WithStatement] with ParenListExpression Substatement

(cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)

Evaluate Validate[ParenListExpression](cxt, env) and ignore its result;

CompileWithFrame: WITHFRAME new WITHFRAME(value: none)
CompileLocalFrame: LOCALFRAME new LOCALFRAME(localBindings: {})
CompileLocalFrame[WithStatement] compileLocalFrame;
CompileEnv: ENVIRONMENT compileLocalFrame ⊕ compileWithFrame ⊕ env;

Evaluate Validate[Substatement](cxt, compileEnv, {}, jt) and ignore its result
end proc;

Setup

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

Evaluation

Eval[WithStatement] with ParenListExpression Substatement

(env: ENVIRONMENT, d: OBJECT): OBJECT

value: OBJECT readReference(Eval[ParenListExpression](env, run), run);

runtimeWithFrame: WITHFRAME new WITHFRAME(value: value)
runtimeLocalFrame: LOCALFRAME instantiateLocalFrame(CompileLocalFrame[WithStatement], runtimeWithFrame ⊕ env);
runtimeEnv: ENVIRONMENT runtimeLocalFrame ⊕ runtimeWithFrame ⊕ env;

return Eval[Substatement](runtimeEnv, d)
end proc;

12.12 Continue and Break Statements

Syntax

ContinueStatement

continue

| continue [no line break] Identifier
BreakStatement []
   break
   | break [no line break] Identifier

Validation

proc Validate[ContinueStatement] (jt: JUMPTARGETS)
   [ContinueStatement [] continue] do
      if default [] jt.continueTargets then
         throw a SyntaxError exception — there is no enclosing statement to which to continue
         end if;
   end proc;

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

Setup

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

proc Setup[BreakStatement] ()
   [BreakStatement [] break] do nothing;
   end proc;

Evaluation

proc Eval[ContinueStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
   [ContinueStatement [] continue] do throw CONTINUE[]value: d, label: default[]
   [ContinueStatement [] continue] do throw CONTINUE[]value: d, label: Name[Identifier]]
   end proc;

   [BreakStatement [] break] do throw BREAK[]value: d, label: default[]
   [BreakStatement [] break] do throw BREAK[]value: d, label: Name[Identifier]]
   end proc;
12.13 Return Statement

Syntax

```
ReturnStatement []
  return
  | return [no line break] ListExpression
```

Validation

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

Setup

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

Evaluation

```
proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
  [ReturnStatement [] return] do throw RETURN[Value: undefined]
  [ReturnStatement [] return [no line break] ListExpression] do
    a: OBJECT [] readReference(Eval[ListExpression]([env, run], run);
    throw RETURN[Value: a]
  end proc;
```

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

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

12.14 Throw Statement

Syntax

```
ThrowStatement [] throw [no line break] ListExpression
```

Validation

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

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

Evaluation

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

12.15 Try Statement

Syntax

TryStatement []
  try Block CatchClauses
  | try Block CatchClausesOpt finally Block

CatchClausesOpt []
  «empty»
  | CatchClauses

CatchClauses []
  CatchClause
  | CatchClauses CatchClause

CatchClause [] catch ( Parameter ) Block

Validation

proc Validate[TryStatement] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP TARGETS)
  [TryStatement [] try Block CatchClauses] do
    Evaluate Validate[Block](ext, env, jt, false) and ignore its result;
    Evaluate Validate[CatchClauses](ext, env, jt) and ignore its result;
  [TryStatement [] try Block1, CatchClausesOpt finally Block2] do
    Evaluate Validate[Block1](ext, env, jt, false) and ignore its result;
    Evaluate Validate[CatchClausesOpt](ext, env, jt) and ignore its result;
    Evaluate Validate[Block2](ext, env, jt, false) and ignore its result
end proc;


Validate[CatchClauses] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP TARGETS) propagates the call to Validate to nonterminals in the expansion of CatchClauses.

CompileEnv[CatchClause]: ENVIRONMENT;

CompileFrame[CatchClause]: LOCAL FRAME;
proc Validate[CatchClause [] catch ( Parameter ) Block] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS)
  compileFrame: LOCAL_FRAME [] new LOCAL_FRAME [localBindings: {}]
  compileEnv: ENVIRONMENT [compileFrame] ⊕ env;
  CompileFrame[CatchClause [] compileFrame;
  CompileEnv[CatchClause [] compileEnv;
  Evaluate Validate[Parameter](ext, compileEnv, compileFrame) and ignore its result;
  Evaluate Validate[Block](ext, compileEnv, jt, false) and ignore its result
end proc;

Setup

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

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

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

proc Setup[CatchClause [] catch ( Parameter ) Block] ()
  Evaluate Setup[Parameter](CompileEnv[CatchClause], CompileFrame[CatchClause], none) and ignore its result;
  Evaluate Setup[Block]() and ignore its result
end proc;

Evaluation

  [TryStatement [] try Block CatchClauses] do
    try return Eval[Block](env, d)
    catch x: SEMANTIC_EXCEPTION do
      if x Œ CONTROLTRANSFER then throw x
      else
        r: OBJECT [] {reject} [] Eval[CatchClauses](env, x);
        if r ≠ reject then return r else throw x end if
      end if
    end try;
TryStatement [] try Block1 CatchClausesOpt finally Block2] do
  result: OBJECTOPT [] none;
  exception: SEMANTICEXCEPTION [] {none} [] none;
  try result [] Eval[Block1](env, d)
catch x: SEMANTICEXCEPTION do exception [] x end try;
  note At this point exactly one of result and exception has a non-none value.
  if exception [] OBJECT then
    try
      r: OBJECT [] {reject} [] Eval[CatchClausesOpt](env, exception);
      if r ≠ reject then
        note The exception has been handled, so clear it.
        result [] r;
        exception [] none
      end if
      catch x: SEMANTICEXCEPTION do
        note The catch clause threw another exception or CONTROLTRANSFER x, so replace the original exception with x.
        exception [] x
      end try
    end if;
  note The finally clause is executed even if the original block exited due to a CONTROLTRANSFER (break, continue, or return).
  note The finally clause is not inside a try-catch semantic statement, so if it throws another exception or CONTROLTRANSFER exception is dropped.
  Evaluate Eval[Block2](env, undefined) and ignore its result;
  note At this point exactly one of result and exception has a non-none value.
  if exception ≠ none then throw exception else return result end if
end proc;

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

proc Eval[CatchClauses](env: ENVIRONMENT, exception: OBJECT): OBJECT [] {reject}
  [CatchClauses [] CatchClause] do return Eval[CatchClause](env, exception);
  [CatchClauses0 [] CatchClauses1 CatchClause] do
    r: OBJECT [] {reject} [] Eval[CatchClauses1](env, exception);
    if r ≠ reject then return r else return Eval[CatchClause](env, exception) end if
end proc;
proc Eval[CatchClause[] catch (Parameter) Block](env: ENVIRONMENT, exception: OBJECT): OBJECT [] {reject}

compileFrame: LOCALFRAME[] CompileFrame[CatchClause];
runtimeFrame: LOCALFRAME[] instantiateLocalFrame(compileFrame, env);
runtimeEnv: ENVIRONMENT[] runtimeFrame @ env;
qname: QUALIFIEDNAME[] public:.(Name[Parameter]);
v: SINGLETONPROPERTYOPT[] findLocalSingletonProperty(runtimeFrame, {qname}, write);

note Validate created one local variable with the name in qname, so v VARIABLE.
if is(exception, v.type) then
  Evaluate writeSingletonProperty(v, exception, run) and ignore its result;
else return Eval[Block](runtimeEnv, undefined)
end if
end proc;

13 Directives

Syntax

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

AnnotatableDirective[]
  VariableDefinition allowSemicolon Semicolon[]
  | FunctionDefinition
  | ClassDefinition
  | NamespaceDefinition Semicolon[]
  | ImportDirective Semicolon[]
  | UseDirective Semicolon[]

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

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

Validation

Enabled[Directive[]]: BOOLEAN;

[Directive[] EmptyStatement] do nothing;
[Directive[] Statement[]] do
  if attr [] {none, true} then
    throw an AttributeError exception — an ordinary statement only permits the attributes true and false
  end if;
end;
Evaluate Validate[Statement[]](ext, env, {}, jt, preinst) and ignore its result;
Evaluate Validate[AnnotatableDirective](ext, env, preinst, attr) and ignore its result;

[Directive][Attributes] do
Evaluate Validate[Attributes](ext, env) and ignore its result;
Evaluate Setup[Attributes]() and ignore its result;
attr2: ATTRIBUTE Eval[Attributes](env, compile);
attr3: ATTRIBUTE combineAttributes(attr, attr2);
if attr3 = false then Enabled[Directive] false
else
   Enabled[Directive] true;
Evaluate Validate[AnnotatableDirective](ext, env, preinst, attr3) and ignore its result
end if;

[Directive][Attributes] do
Evaluate Validate[Attributes](ext, env) and ignore its result;
Evaluate Setup[Attributes]() and ignore its result;
attr2: ATTRIBUTE Eval[Attributes](env, compile);
attr3: ATTRIBUTE combineAttributes(attr, attr2);
if attr3 = false then Enabled[Directive] false
else
   Enabled[Directive] true;
localCxt: CONTEXT new CONTEXT[strict: ext.strict, openNamespaces: ext.openNamespaces]
Evaluate Validate[Directives](localCxt, env, jt, preinst, attr3) and ignore its result
end if;

Pragma Semicolon[] do
if attr[] {none, true} then Evaluate Validate[Pragma](ext) and ignore its result
else
   throw an AttributeError exception — a pragma directive only permits the attributes true and false
end if;
end proc;

proc Validate[AnnotatableDirective]
   (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
[AnnotatableDirective][VariableDefinition] do
Evaluate Validate[VariableDefinition](ext, env, attr) and ignore its result;
[AnnotatableDirective][FunctionDefinition] do
Evaluate Validate[FunctionDefinition](ext, env, preinst, attr) and ignore its result;
[AnnotatableDirective][ClassDefinition] do
Evaluate Validate[ClassDefinition](ext, env, preinst, attr) and ignore its result;
[AnnotatableDirective][NamespaceDefinition] do
Evaluate Validate[NamespaceDefinition](ext, env, preinst, attr) and ignore its result;
Evaluate Validate[ImportDirective](ext, env, preinst, attr) and ignore its result;
   if attr[] {none, true} then
      Evaluate Validate[UseDirective](ext, env) and ignore its result
   else
      throw an AttributeError exception — a use directive only permits the attributes true and false
   end if;
end proc;
Validate[Directives] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN, attr: ATTRIBUTEOptNotFalse) propagates the call to Validate to nonterminals in the expansion of Directives.


Setup

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

proc Setup[AnnotatableDirective] ()
[AnnotatableDirective VariableDefinition Semicolon] do
    Evaluate Setup[VariableDefinition]() and ignore its result;
[AnnotatableDirective FunctionDefinition] do
    Evaluate Setup[FunctionDefinition]() and ignore its result;
[AnnotatableDirective ClassDefinition] do
    Evaluate Setup[ClassDefinition]() and ignore its result;
[AnnotatableDirective NamespaceDefinition Semicolon] do nothing;
end proc;

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

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

Evaluation

[Directive NULL EmptyStatement] do return d;
[Directive Statement] do return Eval[Statement](env, d);
[Directive AnnotatableDirective] do return Eval[AnnotatableDirective](env, d);
    if Enabled[Directive] then return Eval[AnnotatableDirective](env, d)
    else return d
    end if;
[Directives Attributes [no line break] { Directives }] do
    if Enabled[Directive] then return Eval[Directives](env, d) else return d end if;
[Directive] do return d
end proc;

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

[DirectivesPrefix «empty»] do return d;
[DirectivesPrefix0 DirectivesPrefix1 Directive] do
o: OBJECT do Eval[DirectivesPrefix1](env, o);
return Eval[Directive](env, o)end proc;

13.1 Attributes

Syntax

Attributes | Attribute
| AttributeCombination

AttributeCombination | Attribute [no line break] Attributes

Attribute | AttributeExpression
| true
| false
| ReservedNamespace

Validation

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

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

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

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

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

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

Evaluation

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

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

proc Eval[Attribute](env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
    [Attribute [] AttributeExpression] do
        a: OBJECT [] readReference(Eval[AttributeExpression](env, phase), phase);
        return objectToAttribute(a, phase);
    [Attribute [] true] do return true;
    [Attribute [] false] do return false;
    [Attribute [] ReservedNamespace] do return Eval[ReservedNamespace](env, phase)
end proc;

13.2 Use Directive

Syntax

UseDirective [] use namespace ParenListExpression

Validation

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

Syntax

```
importDirective[
  import PackageName
  | import Identifier = PackageName
]
```

Validation

```
proc Validate[importDirective](cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
[importDirective [] import PackageName] do
  if not preinst then
    throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
  end if;
  frame: FRAME [] env[0];
  if frame [] PACKAGE then
    throw a SyntaxError exception — a package may be imported only into a package scope
  end if;
  if attr [] {none, true} then
    throw an AttributeError exception — an unnamed import directive only permits the attributes true and false
  end if;
  pkgName: STRING [] Name[PackageName];
  pkg: PACKAGE [] locatePackage(pkgName);
  Evaluate importPackageInto(pkg, frame) and ignore its result;
[importDirective [] import Identifier = PackageName] do
  if not preinst then
    throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
  end if;
  frame: FRAME [] env[0];
  if frame [] PACKAGE then
    throw a SyntaxError exception — a package may be imported only into a package scope
  end if;
  a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
  if a.implicit then
    throw an AttributeError exception — a package definition cannot have the implicit attribute
  end if;
  if a.prototype then
    throw an AttributeError exception — a package definition cannot have the prototype attribute
  end if;
  pkgName: STRING [] Name[PackageName];
  pkg: PACKAGE [] locatePackage(pkgName);
  Evaluate defineSingletonProperty(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v) and ignore its result;
  Evaluate importPackageInto(pkg, frame) and ignore its result
end proc;
```
proc locatePackage(name: STRING): PACKAGE
    Look for a package bound to name in the implementation’s list of available packages. If one is found, let pkg: PACKAGE be that package; otherwise, throw an implementation-defined error.
    initialize: () (()) {{none, busy}} pkg.initialize;
    case initialize of
      {none} do nothing;
      {busy} do throw an UninitializedError exception — circular package dependency;
      () (()) do
        Evaluate initialize() and ignore its result;
        note pkg.initialize = none;
    end case;
    return pkg
end proc;

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

13.4 Pragma

Syntax

Pragma use Pragmaltems

PragmaItems
  Pragmaltems
  | Pragmaltems

PragmaItem
  PragmaExpr
  | PragmaExpr?

PragmaExpr
  Identifier
  | Identifier (PragmaArgument)

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

Validation

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

Validate[PragmaItems] (ext: CONTEXT) propagates the call to Validate to nonterminals in the expansion of Pragmaltems.
proc Validate[PragmaItem] (ctx: CONTEXT)
    [PragmaItem [] PragmaExpr] do
        Evaluate Validate[PragmaExpr](ctx, false) and ignore its result;
    [PragmaItem [] PragmaExpr ?] do
        Evaluate Validate[PragmaExpr](ctx, true) and ignore its result
end proc;

proc Validate[PragmaExpr] (ctx: CONTEXT, optional: BOOLEAN)
    [PragmaExpr [] Identifier] do
        Evaluate processPragma(ctx, Name[Identifier], undefined, optional) and ignore its result;
    [PragmaExpr [] Identifier (PragmaArgument)] do
        arg: OBJECT Value[PragmaArgument];
        Evaluate processPragma(ctx, Name[Identifier], arg, optional) and ignore its result
end proc;

Value[PragmaArgument]: OBJECT;
Value[PragmaArgument [] true] = true;
Value[PragmaArgument [] false] = false;
Value[PragmaArgument [] Number] = Value[Number];
Value[PragmaArgument [] ~ Number] = generalNumberNegate(Value[Number]);
Value[PragmaArgument [] ~ NegatedMinLong] = (-2^63)_long;
Value[PragmaArgument [] String] = Value[String];

proc processPragma(ctx: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN)
    if name = "strict" then
        if value [] {true, undefined} then ctx.strict [] true; return end if;
        if value = false then ctx.strict [] false; return end if
    end if;
    if name = "ecmascript" then
        if value [] {undefined, 4f64} then return end if;
        if value [] {1f64, 2f64, 3f64} then
            An implementation may optionally modify ctx to disable features not available in ECMAScript Edition value other than subsequent pragmas.
            return
        end if
    end if;
    if not optional then throw a SyntaxError exception end if
end proc;

14 Definitions

14.1 Variable Definition

Syntax

VariableDefinition⁰ [] VariableDefinitionKind VariableBindingList⁰

VariableDefinitionKind []

var
| const
VariableBindingList⁠[\[]⁠:
| VariableBinding⁠[\[]⁠:
  | VariableDefinitionKind⁠ VariableBindingList⁠, VariableBinding⁠[\[]⁠:
VariableBinding⁠[\[]⁠:
  TypedIdentifier⁠ VariableInitialisation⁠[\[]⁠:
VariableInitialisation⁠[\[]⁠:
  «empty»⁠:
  | = VariableInitializer⁠[\[]⁠:
VariableInitializer⁠[\[]⁠:
  AssignmentExpression⁠:
  | AttributeCombination⁠:
TypedIdentifier⁠[\[]⁠:
  Identifier⁠:
  | Identifier⁠ TypeExpression⁠:

Validation

proc Validate⁠[\[]⁠:
  (ext: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOptNOTFALSE)
  Evaluate Validate⁠[\[]⁠(ext, env, attr, Immutable⁠[\[]⁠, false) and ignore its result
end proc;

Immutable⁠[\[]⁠:
  Immutable⁠[\[]⁠:
  Immutable⁠[\[]⁠:
Validate⁠[\[]⁠:
  Validate⁠[\[]⁠(ext, env, attr, Immutable⁠[\[]⁠, false) propagates the call to Validate to nonterminals in the expansion of VariableBindingList⁠[\[]⁠:

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

CompileVar[VariableBinding^0] ▶ v

end case

end if

end proc;

Validate[VariableInitialisation^0](ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of VariableInitialisation^0.

Validate[VariableInitializer^0](ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to nonterminals in the expansion of VariableInitializer^0.

Name[TypedIdentifier^0]: STRING;

Name[TypedIdentifier^0] Identifier = Name[Identifier];

Name[TypedIdentifier^0] Identifier : TypeExpression^0] = Name[Identifier];

Plain[TypedIdentifier^0]: BOOLEAN;

Plain[TypedIdentifier^0] Identifier = true;

Plain[TypedIdentifier^0] Identifier : TypeExpression^0] = false;

proc Validate[TypedIdentifier^0](ext: CONTEXT, env: ENVIRONMENT)

[TypedIdentifier^0] Identifier do nothing;

[TypedIdentifier^0] Identifier : TypeExpression^0] do Evaluate Validate[TypeExpression^0](ext, env) and ignore its result

end proc;

Setup

proc Setup[VariableDefinition^0] VariableDefinitionKind VariableBindingList^0]( )

Evaluate Setup[VariableBindingList^0]() and ignore its result

end proc;

Setup[VariableBindingList^0]() propagates the call to Setup to nonterminals in the expansion of VariableBindingList^0.
proc Setup[VariableBinding[VariableInitialisation]] ()

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

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

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

Evaluation

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

Eval[VariableBindingList] (env: ENVIRONMENT) propagates the call to Eval to nonterminals in the expansion of VariableBindingList.
proc Eval[VariableInitializer][AssignmentExpression](env: Environment, phase: Phase): Object
  return readReference(Eval[AssignmentExpression](env, phase), phase);
proc Eval[AttributeCombination](env, phase)
proc SetupAndEval[TypedIdentifier^] (env: ENVIRONMENT): CLASSOPT
  [TypedIdentifier[] Identifier] do return none;
  [TypedIdentifier[] Identifier : TypeExpression^] do
    return SetupAndEval[TypeExpression^](env)
end proc;

14.2 Simple Variable Definition

Syntax

A SimpleVariableDefinition represents the subset of VariableDefinition expansions that may be used when the variable definition is used as a Substatement[^] 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] (ext: CONTEXT, env: ENVIRONMENT)
  if ext.strict or getRegionalFrame(env) PACKAGE PARAMETERFRAME then
    throw a SyntaxError exception — a variable may not be defined in a substatement except inside a non-strict function or non-strict top-level code; to fix this error, place the definition inside a block
  end if;
  Evaluate Validate[UntypedVariableBindingList](ext, env) and ignore its result
end proc;

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

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

Setup

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

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

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

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

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

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

14.3 Function Definition

Syntax

```plaintext
FunctionDefinition [] function FunctionName FunctionCommon

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

FunctionCommon [] (Parameters) Result Block
```

Validation

```plaintext
OverriddenProperty[FunctionDefinition]: INSTANCETEMPLATEOPT;
```
name: STRING[] Name[FunctionName];
handling: HANDLING[] Handling[FunctionName];
case handling of
{normal} do
kind: STATICFUNCTIONKIND;
if unchecked then kind [] uncheckedFunction
elsif a.prototype then kind [] prototypeFunction
else kind [] plainFunction
end if;
f: SIMPLEINSTANCE[] UNINSTANTIATEDFUNCTION[]
  ValidateStaticFunction[FunctionCommon](cxt, env, kind);
  if preinst then / instantiateFunction(f, env) end if;
  if hoisted then Evaluate defineHoistedVar(env, name, f) and ignore its result
  else
    v: VARIABLE[] new VARIABLE[] type: Function, value: f, immutable: true, setup: none, initializer: none[]
    Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v) and ignore its result
  end if;
{get, set} do
if a.prototype then
  throw an AttributeError exception — a getter or setter cannot have the prototype attribute
end if;
note not (unchecked or hoisted);
Evaluate Validate[FunctionCommon](cxt, env, plainFunction, handling) and ignore its result;
boundEnv: ENVIRONMENTOPT [] none;
if preinst then boundEnv [] env end if;
case handling of
{get} do
  getter: GETTER[] new GETTER[] call: EvalStaticGet[FunctionCommon], env: boundEnv[]
  Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, read, getter)
  and ignore its result;
{set} do
  setter: SETTER[] new SETTER[] call: EvalStaticSet[FunctionCommon], env: boundEnv[]
  Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, write, setter)
  and ignore its result
end case
OverriddenProperty[FunctionDefinition] [] none
end proc;
proc ValidateInstance[FunctionDefinition [] function FunctionName FunctionCommon]
  (ext: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDFUNCTION, final: BOOLEAN)
  if a.prototype then
    throw an AttributeError exception — an instance method cannot have the prototype attribute
  end if;
  handling: HANDLING [] Handling[FunctionName];
  Evaluate Validate[FunctionCommon](ext, env, instanceFunction, handling) and ignore its result;
  signature: PARAMETERFRAME [] CompileFrame[FunctionCommon];
  m: INSTANCEDCRIPTION;
  case handling of
    {normal} do
      m [] new INSTANCEDCRIPTION[final: final, signature: signature, length: signatureLength(signature),
        call: EvalInstanceCall[FunctionCommon]]
    {get} do
      m [] new INSTANCEGETTER[final: final, signature: signature, call: EvalInstanceGet[FunctionCommon]]
    {set} do
      m [] new INSTANCESETTER[final: final, signature: signature, call: EvalInstanceSet[FunctionCommon]]
  end case;
  mOverridden: INSTANCEDCRIPTIONOPT [] defineInstanceProperty(c, ext, Name[FunctionName], a.namespaces,
    a.overrideMod, a.explicit, m);
  enumerable: BOOLEAN [] a.enumerable;
  if mOverridden != none and mOverridden.enumerable then enumerable [] true end if;
  m.enumerable [] enumerable;
  OverriddenProperty[FunctionDefinition] [] mOverridden
end proc;

proc ValidateConstructor[FunctionDefinition [] function FunctionName FunctionCommon]
  (ext: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDFUNCTION)
  if a.prototype then
    throw an AttributeError exception — a class constructor cannot have the prototype attribute
  end if;
  if Handling[FunctionName] [] {get, set} then
    throw a SyntaxError exception — a class constructor cannot be a getter or a setter
  end if;
  Evaluate Validate[FunctionCommon](ext, env, constructorFunction, normal) and ignore its result;
  if c.init != none then
    throw a DefinitionError exception — duplicate constructor definition
  end if;
  c.init [] EvalInstanceInit[FunctionCommon];
  OverriddenProperty[FunctionDefinition] [] none
end proc;
proc Validate[FunctionDefinition [] function FunctionName FunctionCommon]
  (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
  a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
  if a.dyanmic then
    throw an AttributeError exception — a function cannot have the dynamic attribute
  end if;
  frame: FRAME [] env[0];
  if frame [] CLASS then
    note preinst;
  case a.category of
    {static} do
      Evaluate ValidateStatic[FunctionDefinition](ext, env, preinst, a, false, false) and ignore its result;
    {none} do
      if Name[FunctionName] = frame.name then
        Evaluate ValidateConstructor[FunctionDefinition](ext, env, frame, a) and ignore its result
      else
        Evaluate ValidateInstance[FunctionDefinition](ext, env, frame, a, false) and ignore its result
      end if;
    {virtual} do
      Evaluate ValidateInstance[FunctionDefinition](ext, env, frame, a, false) and ignore its result;
    {final} do
      Evaluate ValidateInstance[FunctionDefinition](ext, env, frame, a, true) and ignore its result
  end case
  else
    if a.category ≠ none then
      throw an AttributeError exception — non-class functions cannot have a static, virtual, or final attribute
    end if;
    unchecked: BOOLEAN [] not ext.strict and Handling[FunctionName] = normal and Plain[FunctionCommon];
    hoisted: BOOLEAN [] unchecked and attr = none and
      (frame [] PACKAGE or (frame [] LOCALFRAME and env[1] [] PARAMETERFRAME));
  Evaluate ValidateStatic[FunctionDefinition](ext, env, preinst, a, unchecked, hoisted) and ignore its result
end proc;

Handling[FunctionName]: HANDLING;
  Handling[FunctionName [] Identifier] = normal;
  Handling[FunctionName [] get [no line break] Identifier] = get;
  Handling[FunctionName [] set [no line break] Identifier] = set;

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

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

CompileEnv[FunctionCommon]: ENVIRONMENT;

CompileFrame[FunctionCommon]: PARAMETERFRAME;
proc Validate[FunctionCommon [] (Parameters) Result Block] (cxt: CONTEXT, env: ENVIRONMENT, kind: FUNCTIONKIND, handling: HANDLING)
localCxt: CONTEXT new CONTEXT [strict: cxt.pipeStrict, openNamespaces: cxt.pipeOpenNamespaces]
superconstructorCalled: BOOLEAN [kind ≠ constructorFunction];
compileFrame: PARAMETERFRAME new PARAMETERFRAME [localBindings: {}, kind: kind, handling: handling,
callsSuperconstructor: false, superconstructorCalled: superconstructorCalled, this: none, parameters: [],
rest: none]
compileEnv: ENVIRONMENT [compileFrame] ⊗ env;
CompileFrame[FunctionCommon] [] compileFrame;
CompileEnv[FunctionCommon] [] compileEnv;
if kind = uncheckedFunction then
  Evaluate defineHoistedVar(compileEnv, “arguments”, undefined) and ignore its result
end if;
Evaluate Validate[Parameters](localCxt, compileEnv, compileFrame) and ignore its result;
Evaluate Validate[Result](localCxt, compileEnv) and ignore its result;
Evaluate Validate[Block](localCxt, compileEnv, JUMPTARGETS.breakTargets: {}, continueTargets: {}][false] and ignore its result
end proc;

proc ValidateStaticFunction[FunctionCommon [] (Parameters) Result Block] (cxt: CONTEXT, env: ENVIRONMENT, kind: STATICFUNCTIONKIND): UNINSTANTIATEDFUNCTION
Evaluate Validate[FunctionCommon](ext, env, kind, normal) and ignore its result;
length: INTEGER [] ParameterCount[Parameters];
case kind of
  {plainFunction} do
    return new UNINSTANTIATEDFUNCTION[type: Function, length: length,
call: EvalStaticCall[FunctionCommon], construct: none, instantiations: {}]
  {uncheckedFunction, prototypeFunction} do
    return new UNINSTANTIATEDFUNCTION[type: PrototypeFunction, length: length,
call: EvalStaticCall[FunctionCommon], construct: EvalPrototypeConstruct[FunctionCommon],
instantiations: {}]
end case
end proc;
Setup

\[
\text{proc } \text{Setup}(\text{FunctionDefinition} [] \text{ function } \text{FunctionName} \text{ FunctionCommon})() \\
\text{overriddenProperty: InstancePropertyOpt [] OverriddenProperty[FunctionDefinition];} \\
\text{case overriddenProperty of} \\
\{\text{none} \} \text{ do Evaluate } \text{Setup}(\text{FunctionCommon})() \text{ and ignore its result;} \\
\text{INSTANCEMETHOD [] INSTANCEGETTER [] INSTANCESETTER do} \\
\text{Evaluate } \text{SetupOverride}(\text{FunctionCommon})(\text{overriddenProperty}\cdot\text{signature}) \text{ and ignore its result;} \\
\text{INSTANCEVARIABLE do} \\
\text{overriddenSignature: ParameterFrame;} \\
\text{case Handling[\text{FunctionName}] of} \\
\{\text{normal} \} \text{ do} \\
\text{This cannot happen because ValidateInstance already ensured that a function cannot override an} \\
\text{instance variable.} \\
\{\text{get} \} \text{ do} \\
\text{overriddenSignature [] new ParameterFrame[localBindings: {}}, \text{kind: instanceFunction,} \\
\text{handling: get, callsSuperconstructor: false, superconstructorCalled: false, this: none,} \\
\text{parameters: []}, \text{rest: none, returnType: overriddenProperty.type[]} \\
\{\text{set} \} \text{ do} \\
\forall: \text{Variable [] new Variable[type: overriddenProperty.type, value: none, immutable: false,} \\
\text{setup: none, initializer: none[]} \\
\text{parameters: Parameter[]} [] \text{[Parameter[]} \text{var: \forall, default: none[]];} \\
\text{overriddenSignature [] new ParameterFrame[localBindings: {}}, \text{kind: instanceFunction,} \\
\text{handling: set, callsSuperconstructor: false, superconstructorCalled: false, this: none,} \\
\text{parameters: parameters, rest: none, returnType: Void[]} \\
\text{end case;} \\
\text{Evaluate } \text{SetupOverride}(\text{FunctionCommon})(\text{overriddenSignature}) \text{ and ignore its result} \\
\text{end case} \\
\text{end proc;} \\
\text{proc } \text{Setup}(\text{FunctionCommon} [] \text{ Parameters } \text{ Result Block})() \\
\text{compileEnv: Environment [] CompileEnv[FunctionCommon];} \\
\text{compileFrame: ParameterFrame [] CompileFrame[FunctionCommon];} \\
\text{Evaluate } \text{Setup}(\text{Parameters})(\text{compileEnv, compileFrame}) \text{ and ignore its result;} \\
\text{Evaluate checkAccessorParameters(compileFrame) and ignore its result;} \\
\text{Evaluate } \text{Setup}[\text{Result}](\text{compileEnv, compileFrame}) \text{ and ignore its result;} \\
\text{Evaluate } \text{Setup}[\text{Block}]() \text{ and ignore its result} \\
\text{end proc;} \\
\text{proc } \text{SetupOverride}(\text{FunctionCommon} [] \text{ Parameters } \text{ Result Block})(\text{overriddenSignature: ParameterFrame}) \\
\text{compileEnv: Environment [] CompileEnv[FunctionCommon];} \\
\text{compileFrame: ParameterFrame [] CompileFrame[FunctionCommon];} \\
\text{Evaluate } \text{SetupOverride}(\text{Parameters})(\text{compileEnv, compileFrame, overriddenSignature}) \text{ and ignore its result;} \\
\text{Evaluate checkAccessorParameters(compileFrame) and ignore its result;} \\
\text{Evaluate } \text{SetupOverride}[\text{Result}](\text{compileEnv, compileFrame, overriddenSignature}) \text{ and ignore its result;} \\
\text{Evaluate } \text{Setup}[\text{Block}]() \text{ and ignore its result} \\
\text{end proc;
Evaluation

**proc EvalStaticCall[FunctionCommon] (Parameters) Result Block**

```plaintext
(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
```

*note* The check that `phase ≠ compile` also ensures that `Setup` has been called.

```plaintext
if phase = compile then
  throw a ConstantError exception — a constant expression cannot call user-defined functions
end if;
```

```plaintext
runtimeEnv: ENVIRONMENT f.env;
runtimeThis: OBJECTOPT none;
compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
```

```plaintext
if compileFrame.kind ∈ {uncheckedFunction, prototypeFunction} then
  if this ∈ PRIMITIVEOBJECT then runtimeThis ⊕ getPackageFrame(runtimeEnv)
  else runtimeThis ⊕ this
end if;
```

```plaintext
end if;
```

```plaintext
runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
```

Evaluate `assignArguments(runtimeFrame, f, args, phase)` and ignore its result;

```plaintext
result: OBJECT;
```

try
  Evaluate `Eval[Block](runtimeFrame ⊕ runtimeEnv, undefined)` and ignore its result;
  `result ⊕ undefined`
```plaintext
catch x: SEMANTICEXCEPTION do
  if x ∈ RETURN then result ⊕ x.value else throw x end if
end try;
```

```plaintext
return coerce(result, runtimeFrame.returnType)
```

end proc;

**proc EvalStaticGet[FunctionCommon] (Parameters) Result Block**

```plaintext
(runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
```

*note* The check that `phase ≠ compile` also ensures that `Setup` has been called.

```plaintext
if phase = compile then
  throw a ConstantError exception — a constant expression cannot call user-defined getters
end if;
```

```plaintext
compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
```

```plaintext
runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, none);
```

Evaluate `assignArguments(runtimeFrame, none, [], phase)` and ignore its result;

```plaintext
result: OBJECT;
```

try
  Evaluate `Eval[Block](runtimeFrame ⊕ runtimeEnv, undefined)` and ignore its result;
  `throw a SyntaxError exception — a getter must return a value and may not return by falling off the end of its code`
```plaintext
catch x: SEMANTICEXCEPTION do
  if x ∈ RETURN then result ⊕ x.value else throw x end if
end try;
```

```plaintext
return coerce(result, runtimeFrame.returnType)
```

end proc;
proc `EvalStaticSet`[`FunctionCommon`[] (Parameters) Result Block]
(newValue: OBJECT, runtimeEnv: ENVIRONMENT, phase: PHASE)

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

proc `EvalInstanceCall`[`FunctionCommon`[] (Parameters) Result Block]
(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT

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

proc `EvalInstanceGet`[`FunctionCommon`[] (Parameters) Result Block] (this: OBJECT, phase: PHASE): OBJECT

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

  note  The check that phase ≠ compile also ensures that Setup has been called.
  if phase = compile then
    throw a ConstantError exception — a constant expression cannot call setters
  end if;

  note  Class frames are always preinstantiated, so the run environment is the same as compile environment.
  env: ENVIRONMENT [] CompileEnv[FunctionCommon];
  compileFrame: PARAMETERFRAME [] CompileFrame[FunctionCommon];
  runtimeFrame: PARAMETERFRAME [] instantiateParameterFrame(compileFrame, env, this);

  Evaluate assignArguments(runtimeFrame, none, [newValue], phase) and ignore its result;
  try Evaluate Eval[Block](runtimeFrame ⊕ env, undefined) and ignore its result
  catch x: SEMANTICEXCEPTION do if x œ RETURN then throw x end if
  end try
end proc;

proc EvalInstanceInit[FunctionCommon [] (Parameters) Result Block]
  (this: SIMPLEINSTANCE, args: OBJECT[][], phase: {run})

  note  Class frames are always preinstantiated, so the run environment is the same as compile environment.
  env: ENVIRONMENT [] CompileEnv[FunctionCommon];
  compileFrame: PARAMETERFRAME [] CompileFrame[FunctionCommon];
  runtimeFrame: PARAMETERFRAME [] instantiateParameterFrame(compileFrame, env, this);

  Evaluate assignArguments(runtimeFrame, none, args, phase) and ignore its result;
  if not runtimeFrame.callsSuperconstructor then
    c: CLASS [] getEnclosingClass(env);
    Evaluate callInit(this, c.super, [], run) and ignore its result;
    runtimeFrame.superconstructorCalled [] true
  end if;

  try Evaluate Eval[Block](runtimeFrame ⊕ env, undefined) and ignore its result
  catch x: SEMANTICEXCEPTION do if x œ RETURN then throw x end if
  end try;

  if not runtimeFrame.superconstructorCalled then
    throw an UninitializedError exception — the superconstructor must be called before returning normally from a
    constructor
  end if
end proc;
proc评VariantPrototypeConstruct[FunctionCommon Ren 参数 Result Block]
   (f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT

note The check that phase ≠ compile also ensures that Setup has been called.
if phase = compile then
   throw a ConstantError exception — a constant expression cannot call user-defined prototype constructors
end if;

runtimeEnv: ENVIRONMENT f.env;
archetype: OBJECT dotRead(f, {public: "prototype"}, phase);
if archetype ∈ {null, undefined} then archetype = ObjectPrototype
elsif objectType(archetype) ≠ Object then
   throw a TypeError exception — bad prototype value
end if;
o: OBJECT createSimpleInstance(Object, archetype, none, none, none);
compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, o);
Evaluate assignArguments(runtimeFrame, f, args, phase) and ignore its result;
result = OBJECT;
try
   Evaluate Eval[Block](runtimeFrame ⊕ runtimeEnv, undefined) and ignore its result;
result = undefined
catch x: SEMANTICEXCEPTION do
   if x ∈ RETURN then result = x.value else throw x end if
end try;
coercedResult = OBJECT coerce(result, runtimeFrame.returnType);
if coercedResult ∈ PRIMITIVEOBJECT then return o else return coercedResult end if
end proc;

proc checkAccessorParameters(frame: PARAMETERFRAME)
   parameters: PARAMETER[] frame.parameters;
   rest: VARIABLEOPT frame.rest;
case frame.handling of
   {normal} do nothing;
   {get} do
      if parameters ≠ [] or rest ≠ none then
         throw a SyntaxError exception — a getter cannot take any parameters
      end if;
   {set} do
      if |parameters| ≠ 1 or rest ≠ none then
         throw a SyntaxError exception — a setter must take exactly one parameter
      end if;
      if parameters[0].default ≠ none then
         throw a SyntaxError exception — a setter’s parameter cannot be optional
      end if
   end case
end proc;
proc assignArguments(runtimeFrame: PARAMETERFRAME, f: SIMPLEINSTANCE [] {none}, args: OBJECT[])

This procedure performs a number of checks on the arguments, including checking their count, names, and values.
Although this procedure performs these checks in a specific order for expository purposes, an implementation may perform these checks in a different order, which could have the effect of reporting a different error if there are multiple errors. For example, if a function only allows between 2 and 4 arguments, the first of which must be a Number and is passed five arguments the first of which is a String, then the implementation may throw an exception either about the argument count mismatch or about the type coercion error in the first argument.

argumentsObject: OBJECTOPT [] none;

if runtimeFrame.kind = uncheckedFunction then
    argumentsObject [] construct(Array, [], phase);
    Evaluate createDynamicProperty(argumentsObject, public::"callee", false, false, f) and ignore its result;
    Evaluate writeArrayPrivateLength(argumentsObject, |args|, phase) and ignore its result
end if;

restObject: OBJECTOPT [] none;
rest: VARIABLE [] {none} [] runtimeFrame.rest;
if rest ≠ none then restObject [] construct(Array, [], phase) end if;

parameters: PARAMETER [] runtimeFrame.parameters;
if i: INTEGER 0;
j: INTEGER 0;
for each arg [] args do
    if i < |parameters| then
        parameter: PARAMETER [] parameters[i];
        default: OBJECTOPT [] parameter.default;
        argOrDefault: OBJECT [] arg;
        if argOrDefault = undefined and default ≠ none then argOrDefault [] default
    end if;
    v: DYNAMICVAR VARIABLE [] parameter.var;
    Evaluate writeSingletonProperty(v, argOrDefault, phase) and ignore its result;
    if argumentsObject ≠ none then
        note Create an alias of v as the ith entry of the arguments object.
        note v DYNAMICVAR;
        qname: QUALIFIEDNAME [] objectToQualifiedName(i, phase);
        argumentsObject.localBindings [] argumentsObject.localBindings [] {LOCALBINDING}name: qname,
        accesses: readWrite, explicit: false, enumerable: false, content: f
    end if
    elsif restObject ≠ none then
        if j ≥ arrayLimit then throw a RangeError exception end if;
        Evaluate indexWrite(restObject, j, arg, phase) and ignore its result;
        note argumentsObject = none because a function can't have both a rest parameter and an arguments object.
        j [] j + 1
    elsif argumentsObject ≠ none then
        Evaluate indexWrite(argumentsObject, i, arg, phase) and ignore its result
    else
        throw an ArgumentError exception — more arguments than parameters were supplied, and the called function
does not have a . . . parameter and is not unchecked.
    end if;
    i [] i + 1
end for each;
while i < |parameters| do
    parameter: PARAMETER [] parameters[i];
    default: OBJECTOPT [] parameter.default;
    if default = none then
        if argumentsObject ≠ none then default [] undefined
    else
throw an ArgumentError exception — fewer arguments than parameters were supplied, and the called function does not supply default values for the missing parameters and is not unchecked.

eend if
eend if;

Evaluate writeSingletonProperty( parameter.var, default, phase ) and ignore its result;

i = i + 1
end while
eend proc;

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

Syntax

Parameters □
  «empty»
  | NonemptyParameters

NonemptyParameters □
  ParameterInit
  | ParameterInit, NonemptyParameters
  | RestParameter

Parameter □ ParameterAttributes TypedIdentifier

ParameterAttributes □
  «empty»
  | const

ParameterInit □
  Parameter
  | Parameter = AssignmentExpression

RestParameter □

  ...
  | ... ParameterAttributes Identifier

Result □
  «empty»
  | : TypeExpression

Validation

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

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

Validate[Parameters] (ext: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME) propagates the call to Validate to nonterminals in the expansion of Parameters.
Plain[NonemptyParameters]: BOOLEAN;
Plain[NonemptyParameters [] ParameterInit] = Plain[ParameterInit];
Plain[NonemptyParameters0 [] ParameterInit , NonemptyParameters1]
  = Plain[ParameterInit] and Plain[NonemptyParameters1];
Plain[NonemptyParameters [] RestParameter] = false;

ParameterCount[NonemptyParameters]: INTEGER;
ParameterCount[NonemptyParameters [] ParameterInit] = 1;
ParameterCount[NonemptyParameters0 [] ParameterInit , NonemptyParameters1]
  = 1 + ParameterCount[NonemptyParameters1];
ParameterCount[NonemptyParameters [] RestParameter] = 0;

Validate[NonemptyParameters] (ext: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME) propagates the
call to Validate to nonterminals in the expansion of NonemptyParameters.

Name[Parameter [] ParameterAttributes TypedIdentifierallowIn]: STRING = Name[TypedIdentifierallowIn];

Plain[Parameter [] ParameterAttributes TypedIdentifierallowIn]: BOOLEAN
  = Plain[TypedIdentifierallowIn] and not HasConst[ParameterAttributes];

CompileVar[Parameter]: DYNAMICVAR [] VARIABLE;

proc Validate[Parameter [] ParameterAttributes TypedIdentifierallowIn]
  (ext: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME [], LOCALFRAME)
Evaluate Validate[TypedIdentifierallowIn](ext, env) and ignore its result;
immutable: BOOLEAN [] HasConst[ParameterAttributes];
name: STRING [] Name[TypedIdentifierallowIn];
v: DYNAMICVAR [] VARIABLE;
if compileFrame [] PARAMETERFRAME and compileFrame.kind = uncheckedFunction then
  note not immutable;
  v [] defineHoistedVar(env, name, undefined)
else
  v [] new VARIABLE[value: none, immutable: immutable, setup: none, initializer: none]
  Evaluate defineSingletonProperty(env, name, {public}, none, false, readWrite, v) and ignore its result
end if;
CompileVar[Parameter] [] v
end proc;

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

Plain[ParameterInit]: BOOLEAN;
Plain[ParameterInit [] Parameter] = Plain[Parameter];
Plain[ParameterInit [] Parameter = AssignmentExpressionallowIn] = false;

proc Validate[ParameterInit] (ext: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[ParameterInit [] Parameter] do
  Evaluate Validate[Parameter](ext, env, compileFrame) and ignore its result;
[ParameterInit [] Parameter = AssignmentExpressionallowIn] do
  Evaluate Validate[Parameter](ext, env, compileFrame) and ignore its result;
  Evaluate Validate[AssignmentExpressionallowIn](ext, env) and ignore its result
end proc;
proc Validate[RestParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME) ...

Plain[Result]: BOOLEAN;
Plain[Result «empty»] = true;
Plain[Result : TypeExpression"low"] = false;

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

Setup

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

proc SetupOverride[Parameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenSignature: PARAMETERFRAME) ...

[NonemptyParameters «empty»] do
if overriddenSignature.parameters ≠ [] or overriddenSignature.rest ≠ none then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
end if;
[NonemptyParameters NonemptyParameters] do
Evaluate SetupOverride[NonemptyParameters](compileEnv, compileFrame, overriddenSignature, overriddenSignature.parameters) and ignore its result
end proc;

proc Setup[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME) ...

[NonemptyParameters ParameterInit] do
Evaluate Setup[ParameterInit](compileEnv, compileFrame) and ignore its result;
[NonemptyParameters ParameterInit NonemptyParameters] do
Evaluate Setup[ParameterInit](compileEnv, compileFrame) and ignore its result;
Evaluate Setup[NonemptyParameters](compileEnv, compileFrame) and ignore its result;
[NonemptyParameters RestParameter] do nothing
end proc;
proc SetupOverride[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME,
overriddenSignature: PARAMETERFRAME, overriddenParameters: PARAMETER[]) [NonemptyParameters [] ParameterInit] do
  if overriddenParameters = [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  Evaluate SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]) and ignore its result;
if |overriddenParameters| ≠ 1 or overriddenSignature.rest ≠ none then
  throw a DefinitionError exception — mismatch with the overridden method’s signature
end if;
[NonemptyParameters] [NonemptyParameters] do
  if overriddenParameters = [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  Evaluate SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]) and ignore its result;
  Evaluate SetupOverride[NonemptyParameters](compileEnv, compileFrame, overriddenSignature, overriddenParameters[1 ...]) and ignore its result;
[NonemptyParameters] RestParameter do
  if overriddenParameters ≠ [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  overriddenRest: VARIABLE {none} overriddenSignature.rest;
if overriddenRest = none or overriddenRest.type ≠ Array then
  throw a DefinitionError exception — mismatch with the overridden method’s signature
end if
end proc;

proc Setup[Parameter [] ParameterAttributes TypedIdentifier allowIn] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME [] LOCALFRAME, default: OBJECTOPT)
if compileFrame: PARAMETERFRAME and default = none and
  (some p2: compileFrame.parameters satisfies p2.default ≠ none) then
  throw a SyntaxError exception — a required parameter cannot follow an optional one
end if;
v: DYNAMICVAR VARIABLE CompileVar[Parameter];
case v of
  DYNAMICVAR do nothing;
  VARIABLE do
    type: CLASSOPT _ SetupAndEval[TypedIdentifier allowIn](compileEnv);
    if type = none then type: Object end if;
    v.type [] type
  end case;
if compileFrame: PARAMETERFRAME then
end if
end proc;
proc SetupOverride[Parameter [] ParameterAttributes TypedIdentifier allowIn] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, default: OBJECTOPT, overriddenParameter: PARAMETER)
newDefault: OBJECTOPT [] default;
if newDefault = none then newDefault [] overriddenParameter.default end if;
if default = none and (some p2 [] compileFrame.parameters satisfies p2.default ≠ none) then
throw a SyntaxError exception — a required parameter cannot follow an optional one
end if;
v: DYNAMICVAR VARIABLE CompileVar[Parameter];
note v [] DYNAMICVAR;
type: CLASSOPT SetupAndEval[TypedIdentifier allowIn](compileEnv);
if type = none then type Object end if;
if type ≠ overriddenParameter.var.type then
throw a DefinitionError exception — mismatch with the overridden method’s signature
end if;
v.type [] type;
p: PARAMETER PARAMETER [var: v, default: newDefault[]]
compileFrame.parameters [] compileFrame.parameters ⊕ [p]
ext proc;

proc Setup[ParameterInit] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[ParameterInit [] Parameter] do
Evaluate Setup[Parameter](compileEnv, compileFrame, none) and ignore its result;
[ParameterInit [] Parameter = AssignmentExpression allowIn] do
Evaluate Setup[AssignmentExpression allowIn]() and ignore its result;
default: OBJECT [] readReference(Eval[AssignmentExpression allowIn](compileEnv, compile), compile);
Evaluate Setup[Parameter](compileEnv, compileFrame, default) and ignore its result
end proc;

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

proc Setup[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[Result [] «empty»] do
defaultReturnType: CLASS Object;
if cannotReturnValue(compileFrame) then defaultReturnType [] Void end if;
compileFrame.returnType [] defaultReturnType;
[Result [] : TypeExpression allowIn] do
if cannotReturnValue(compileFrame) then
throw a SyntaxError exception — a setter or constructor cannot define a return type
end if;
compileFrame.returnType [] SetupAndEval[TypeExpression allowIn](compileEnv)
ext proc;
proc SetupOverride(Result) (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenSignature: PARAMETERFRAME)
    [Result [ «empty»] do compileFrame.returnType [ overriddenSignature.returnType; 
    [Result [ : TypeExpression allowIn] do
        t: CLASS [] SetupAndEval[TypeExpression allowIn](compileEnv); 
        if overriddenSignature.returnType ≠ t then
            throw a DefinitionError exception — mismatch with the overridden method’s signature
        end if;
        compileFrame.returnType [] t
    end proc;

14.4 Class Definition

Syntax

ClassDefinition [] class Identifier Inheritance Block

Inheritance []
    «empty»
    | extends TypeExpression allowIn

Validation

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

proc Validate[Inheritance] (ext: CONTEXT, env: ENVIRONMENT): CLASS
  [Inheritance [] «empty»] do return Object;
  [Inheritance [] extends TypeExpression[allow]] do
    Evaluate Validate[TypeExpression[allow]](ext, env) and ignore its result;
    return SetupAndEval[TypeExpression[allow]](env)
  end proc;
Setup

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

Evaluation

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

14.5 Namespace Definition

Syntax

```
NamespaceDefinition [] namespace Identifier
```

Validation

```
proc Validate[NamespaceDefinition [] namespace Identifier]
    (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOptNotFalse)
    if not preinst then
        throw a SyntaxError exception — a namespace may be defined only in a preinstantiated scope
    end if;
    a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
    if a.dynamic then
        throw an AttributeError exception — a namespace definition cannot have the dynamic attribute
    end if;
    if a.prototype then
        throw an AttributeError exception — a namespace definition cannot have the prototype attribute
    end if;
    case a.category of
        {none} do nothing;
        {static} do
        if env[0] [] CLASS then
            throw an AttributeError exception — non-class property definitions cannot have a static attribute
        end if;
        {virtual, final} do
            throw an AttributeError exception — a namespace definition cannot have the virtual or final attribute
        end case;
        name: STRING [] Name[Identifier];
    ns: NAMESPACE [] new NAMESPACE[]name: name[]
    v: VARIABLE [] new VARIABLE[]type: Namespace, value: ns, immutable: true, setup: none, initializer: none[]
    Evaluate defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v) and ignore its result
end proc;
```
1615 Programs

Syntax

Program $_$
  Directives
  | PackageDefinition Program

Processing

Process[Program]: OBJECT;
Process[Program $\rightarrow$ Directives]
  begin
    ext: CONTEXT $\rightarrow$ new CONTEXT\strict: false, openNamespaces: \{public, internal\}$\rightarrow$
    initialEnvironment: ENVIRONMENT $\rightarrow$ \{createGlobalObject\};
    Evaluate Validate[Directives](ext, initialEnvironment,
       JUMP\{breakTargets: \{}, continueTargets: \{\}true, none\} and ignore its result;
    Evaluate Setup[Directives]() and ignore its result;
    return Eval[Directives](initialEnvironment, undefined)$\rightarrow$
  end;
Process[Program]$\rightarrow$ PackageDefinition Program$_1$
  begin
    Evaluate Process[PackageDefinition] and ignore its result;
    return Process[Program$_1$]
  end;

15.1 Package Definition

Syntax

PackageDefinition $_\rightarrow$ package PackageNameOpt Block

PackageNameOpt $_\rightarrow$
  «empty»
  | PackageName

PackageName $_\rightarrow$
  String
  | PackageIdentifiers

PackageIdentifiers $_\rightarrow$
  Identifier
  | PackageIdentifiers . Identifier
Processing

**Process**([PackageDefinition] [] **package** PackageNameOpt Block): VOID

begin

  name: STRING [] Name[PackageNameOpt];
  cxt: CONTEXT [] new CONTEXT[strict: false, openNamespaces: {public, internal}]
  globalObject: PACKAGE [] createGlobalObject();
  pkgInternal: NAMESPACE [] new NAMESPACE[name: “internal”]
  pkg: PACKAGE [] new PACKAGE[localBindings:
    {stdExplicitConstBinding(internal::“internal”, Namespace, internal) }, archetype: ObjectPrototype,
    name: name, initialize: busy, sealed: true, internalNamespace: pkgInternal]
  initialEnvironment: ENVIRONMENT [] [pkg, globalObject];
  Evaluate Validate[Block](cxt, initialEnvironment, JUMPTARGETS[breakTargets: {}, continueTargets: {}][true])
    and ignore its result;
  Evaluate Setup[Block]() and ignore its result;
  proc evalPackage()]
    pkg.initialize [] busy;
    Evaluate Eval[Block](initialEnvironment, undefined) and ignore its result;
    pkg.initialize [] none
end proc;
  pkg.initialize [] evalPackage;
  Bind name to package pkg in the system’s list of packages in an implementation-defined manner.
end;

**Name**[PackageNameOpt]: STRING;

**Name**[PackageNameOpt] «empty» = an implementation-supplied name;
**Name**[PackageNameOpt] PackageName = Name[PackageName];

**Name**[PackageName]: STRING;

**Name**[PackageName] String = Value[String] processed in an implementation-defined manner;
**Name**[PackageName] PackageIdentifiers = Names[PackageIdentifiers] processed in an implementation-defined manner;

**Names**[PackageIdentifiers]: STRING[];

**Names**[PackageIdentifiers] Identifier = [Name[Identifier]];
**Names**[PackageIdentifierso] PackageIdentifiers . Identifier = Names[PackageIdentifiers] ⊕ [Name[Identifier]];

packageDatabase: PACKAGE[] [] {};

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

proc createGlobalObject(): PACKAGE
return new PACKAGE{localBindings: {
    stdExplicitConstBinding(internal::“internal”, Namespace, internal),
    stdConstBinding(public::“explicit”, Attribute, global_explicit),
    stdConstBinding(public::“enumerable”, Attribute, global_enumerable),
    stdConstBinding(public::“dynamic”, Attribute, global_dynamic),
    stdConstBinding(public::“static”, Attribute, global_static),
    stdConstBinding(public::“virtual”, Attribute, global_virtual),
    stdConstBinding(public::“final”, Attribute, global_final),
    stdConstBinding(public::“prototype”, Attribute, global_prototype),
    stdConstBinding(public::“unused”, Attribute, global_unused),
    stdFunction(public::“override”, global_override, 1),
    stdConstBinding(public::“NaN”, Number, NaN164),
    stdConstBinding(public::“Infinity”, Number, +0x164),
    stdConstBinding(public::“fNaN”, float, NaN32),
    stdConstBinding(public::“fInfinity”, float, +0x32),
    stdConstBinding(public::“undefined”, Void, undefined),
    stdFunction(public::“eval”, global_eval, 1),
    stdFunction(public::“parseInt”, global_parseint, 2),
    stdFunction(public::“parseLong”, global_parselong, 2),
    stdFunction(public::“parseFloat”, global_parsefloat, 1),
    stdFunction(public::“isNaN”, global_isNaN, 1),
    stdFunction(public::“isFinite”, global_isfinite, 1),
    stdFunction(public::“decodeURI”, global_decodeuri, 1),
    stdFunction(public::“decodeURIComponent”, global_decodeuricomponent, 1),
    stdFunction(public::“encodeURI”, global_encodeuri, 1),
    stdFunction(public::“encodeURIComponent”, global_encodeuricomponent, 1),
    stdConstBinding(public::“Object”, Class, Object),
    stdConstBinding(public::“Never”, Class, Never),
    stdConstBinding(public::“Void”, Class, Void),
    stdConstBinding(public::“Null”, Class, Null),
    stdConstBinding(public::“Boolean”, Class, Boolean),
    stdConstBinding(public::“GeneralNumber”, Class, GeneralNumber),
    stdConstBinding(public::“long”, Class, long),
    stdConstBinding(public::“ulong”, Class, ulong),
    stdConstBinding(public::“float”, Class, float),
    stdConstBinding(public::“Number”, Class, Number),
    stdConstBinding(public::“sbyte”, Class, sbyte),
    stdConstBinding(public::“byte”, Class, byte),
    stdConstBinding(public::“short”, Class, short),
    stdConstBinding(public::“ushort”, Class, ushort),
    stdConstBinding(public::“int”, Class, int),
    stdConstBinding(public::“uint”, Class, uint),
    stdConstBinding(public::“char”, Class, char),
    stdConstBinding(public::“String”, Class, String),
    stdConstBinding(public::“Array”, Class, Array),
    stdConstBinding(public::“Namespace”, Class, Namespace),
    stdConstBinding(public::“Attribute”, Class, Attribute),
    stdConstBinding(public::“Date”, Class, Date),
    stdConstBinding(public::“RegExp”, Class, RegExp),
    stdConstBinding(public::“Class”, Class, Class),
    stdConstBinding(public::“Function”, Class, Function),
    stdConstBinding(public::“PrototypeFunction”, Class, PrototypeFunction),
    stdConstBinding(public::“Package”, Class, Package),
    stdConstBinding(public::“Error”, Class, Error),
}}
stdConstBinding(public::"ArgumentError", Class, ArgumentError),  
stdConstBinding(public::"AttributeError", Class, AttributeError),  
stdConstBinding(public::"ConstantError", Class, ConstantError),  
stdConstBinding(public::"DefinitionError", Class, DefinitionError),  
stdConstBinding(public::"EvalError", Class, EvalError),  
stdConstBinding(public::"RangeError", Class, RangeError),  
stdConstBinding(public::"ReferenceError", Class, ReferenceError),  
stdConstBinding(public::"SyntaxError", Class, SyntaxError),  
stdConstBinding(public::"TypeError", Class, TypeError),  
stdConstBinding(public::"UninitializedError", Class, UninitializedError),  
stdConstBinding(public::"URIError", Class, URIError),
archetype: ObjectPrototype, name: "", initialize: none, sealed: false, internalNamespace: internal
end proc;

16.1 Built-in Namespaces

public: NAMESPACE = new NAMESPACE[name: "public"]

internal: NAMESPACE = new NAMESPACE[name: "internal"]

16.2 Built-in Attributes

global_explicit: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: true, enumerable: false,  
  dynamic: false, category: none, overrideMod: none, prototype: false, unused: false]

globalEnumerable: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: true, dynamic: false, category: none, overrideMod: none, prototype: false, unused: false]

globalDynamic: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: true, category: none, overrideMod: none, prototype: false, unused: false]

globalStatic: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: false, category: static, overrideMod: none, prototype: false, unused: false]

globalVirtual: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: false, category: virtual, overrideMod: none, prototype: false, unused: false]

globalFinal: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: false, category: final, overrideMod: none, prototype: false, unused: false]

globalPrototype: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: false, category: none, overrideMod: none, prototype: true, unused: false]

globalUnused: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE[namespaces: {}, explicit: false,  
  enumerable: false, dynamic: false, category: none, overrideMod: none, prototype: false, unused: true]
proc global_override(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object

note This function does not check phase and therefore can be used in a constant expression.

overrideMod: OVERRIDEMODIFIER;

if args = [] then overrideMod = true
elseif |args| = 1 then
    arg: Object[] arg[0];
    if arg œ {true, false, undefined} then throw a TypeError exception end if;
    overrideMod = arg
else throw an ArgumentError exception — too many arguments supplied
end if;


end proc;

16.3 Built-in Functions

proc global_eval(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object

Evaluate ???? and ignore its result

end proc;

proc global_parseint(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Float64

note This function can be used in a constant expression if the arguments can be converted to primitives in constant expressions.

if |args| œ {1, 2} then
    throw an ArgumentError exception — at least one and at most two arguments must be supplied
end if;

s: String[] objectToString(args[0], phase);
radix: INTEGER[] objectToInteger(defaultArg(args, 1, +zero, +zero), phase);
i: (INTEGER - {0})[] {+zero, -zero, NaN}[] stringPrefixToInteger(s, radix);

return extendedRationalToFloat64(i)

end proc;
proc stringPrefixToInteger(s: STRING, radix: INTEGER): (INTEGER – {0}) [] {+zero, –zero, NaN}
    r: INTEGER [] radix;
    if r [] {0, 2 ... 36} then throw a RangeError exception — radix out of range end if;
    i: INTEGER [] 0;
    while i < |s| and the nonterminal WhiteSpaceOrLineTerminatorChar can expand into [s[i]] do
        i [] i + 1
    end while;
    sign: {−1, 1} [] 1;
    if i < |s| then
        if s[i] = ‘+’ then i [] i + 1 elsif s[i] = ‘−’ then sign [] −1; i [] i + 1 end if
    end if;
    if r [] {0, 16} and i + 2 ≤ |s| and s[i ... i + 1] [] {'0x', ‘0X’} then
        r [] 16;
        i [] i + 2
    end if;
    if r = 0 then r [] 10 end if;
    n: INTEGER [] 0;
    start: INTEGER [] i;
    digit: INTEGEROPT [] 0;
    while i < |s| and digit ≠ none do
        ch: CHAR16 [] s[i];
        if ch [] {'0' ... '9'} then digit [] char16ToInteger(ch) − char16ToInteger('0')
            elsif ch [] {'A' ... 'Z'} then
                digit [] char16ToInteger(ch) − char16ToInteger('A') + 10
                elsif ch [] {'a' ... 'z'} then
                    digit [] char16ToInteger(ch) − char16ToInteger('a') + 10
                else digit [] none
            end if;
        if digit ≠ none and digit ≥ r then digit [] none end if;
        if digit ≠ none then n [] n[]r + digit; i [] i + 1 end if
    end while;
    if i = start then return NaN end if;
    if n ≠ 0 then return n[]sign
    elsif sign > 0 then return +zero
    else return –zero
    end if
end proc;

proc global_parseLong(this: OBJECT, if: SIMPLExINSTANCE, args: OBJECT[], phase: PHASE): GENERALNUMBER
    note This function can be used in a constant expression if the arguments can be converted to primitives in constant
    expressions.
    if args [] {1, 2} then
        throw an ArgumentError exception — at least one and at most two arguments must be supplied
    end if;
    s: STRING [] objectToString(args[0], phase);
    radix: INTEGER [] objectToInteger(defaultArg(args, 1, +zeroI64), phase);
    i: (INTEGER – {0}) [] {+zero, –zero, NaN} [] stringPrefixToInteger(s, radix);
    case i of
        {+zero, –zero} do return 0I64;
        INTEGER do return integerToLong(i);
        {NaN} do return NaNI64
    end case
end proc;
proc global_parsefloat(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Float64
  note This function can be used in a constant expression if its argument can be converted to a primitive in a constant expression.
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  s: STRING [] objectToString(args[0], phase);
  Apply the lexer grammar with the start symbol StringDecimalLiteral to the string s. If the grammar can interpret neither s nor any prefix of s as an expansion of StringDecimalLiteral, then return NaN64. Otherwise, let p be the longest prefix of s (possibly s itself) such that p is an expansion of StringDecimalLiteral.
  q: ExtendedRational [] the value of the action Lex applied to p’s expansion of the nonterminal StringDecimalLiteral;
  return extendedRationalToFloat64(q)
end proc;

proc global_isnan(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Boolean
  note This function can be used in a constant expression if its argument can be converted to a primitive in a constant expression.
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  x: GeneralNumber [] objectToGeneralNumber(args[0], phase);
  return x [] {NaN32, NaN64}
end proc;

proc global_isfinite(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Boolean
  note This function can be used in a constant expression if its argument can be converted to a primitive in a constant expression.
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  x: GeneralNumber [] objectToGeneralNumber(args[0], phase);
  return x [] {NaN32, NaN64, +∞128, +∞164, −∞128, −∞164}
end proc;

proc global_decodeuri(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
  Evaluate ??? and ignore its result
end proc;

proc global_decodeuricomponent(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
  Evaluate ??? and ignore its result
end proc;

proc global_encodeuri(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
  Evaluate ??? and ignore its result
end proc;

proc global_encodeuricomponent(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
  Evaluate ??? and ignore its result
end proc;

17 Built-in Classes

proc dummyCall(this: Object, c: Class, args: Object[], phase: Phase): Object
  Evaluate ??? and ignore its result
end proc;
17.1 Object


proc callObject(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in a constant expression.
  if [args] = 0 then return undefined
  elsif [args] = 1 then return args[0]
  else throw an ArgumentError exception — at most one argument can be supplied
end if
end proc;

proc constructObject(c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in a constant expression.
  if [args] > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  o: OBJECT [] defaultArg(args, 0, undefined);
  if o [] {null, undefined} then
    return createSimpleInstance( Object, ObjectPrototype, none, none, none)
  else return o
end if
end proc;

proc coerceObject(o: OBJECT, c: CLASS): OBJECTOPT
  return o
end proc;

ObjectPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
  stdConstBinding(public::"constructor", Class, Object),
  stdFunction(public::"toString", Object_toString, 0),
  stdFunction(public::"toLocaleString", Object_toLocaleString, 0),
  stdFunction(public::"valueOf", Object_valueOf, 0),
  stdFunction(public::"hasOwnProperty", Object_hasOwnProperty, 1),
  stdFunction(public::"isPrototypeOf", Object_isPrototypeOf, 1),
  stdFunction(public::"propertyIsEnumerable", Object_propertyIsEnumerable, 1),
  stdFunction(public::"sealProperty", Object_sealProperty, 1}),

proc Object_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function does not check phase and therefore can be used in a constant expression.
  note This function ignores any arguments passed to it in args.
  c: CLASS [] objectType(this);
  return “[object " ⊕ c.name ⊕ “]”
end proc;
proc Object_toLocaleString(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
if phase = compile then
  throw a ConstantError exception — toLocaleString cannot be called from a constant expression
end if;
toStringMethod: Object [] dotRead(this, {public::“toString”), phase);
return call(this, toStringMethod, args, phase)
end proc;

proc Object_valueOf(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
note This function does not check phase and therefore can be used in a constant expression.
ote This function ignores any arguments passed to it in args.
return this
end proc;

proc Object_hasOwnProperty(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Boolean
if phase = compile then
  throw a ConstantError exception — hasOwnProperty cannot be called from a constant expression
end if;
if [args] ≠ 1 then
  throw an ArgumentError exception — exactly one argument must be supplied
end if;
return hasProperty(this, args[0], true, phase)
end proc;

proc Object_isPrototypeOf(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Boolean
if phase = compile then
  throw a ConstantError exception — isPrototypeOf cannot be called from a constant expression
end if;
if [args] ≠ 1 then
  throw an ArgumentError exception — exactly one argument must be supplied
end if;
o: Object [] args[0];
return this [] archetypes(o)
end proc;

proc Object_propertyIsEnumerable(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Boolean
if phase = compile then
  throw a ConstantError exception — propertyIsEnumerable cannot be called from a constant expression
end if;
if [args] ≠ 1 then
  throw an ArgumentError exception — exactly one argument must be supplied
end if;
qname: QualifiedName [] objectToQualifiedName(args[0], phase);
c: Class [] objectType(this);
mBase: InstancePropertyOpt [] findBaseInstanceProperty(c, {qname}, read);
if mBase ≠ none then
  m: InstanceProperty [] getDerivedInstanceProperty(c, mBase, read);
  if m.enumerable then return true end if
end if;
mBase [] findBaseInstanceProperty(c, {qname}, write);
if mBase ≠ none then
  m: InstanceProperty [] getDerivedInstanceProperty(c, mBase, write);
  if m.enumerable then return true end if
end if;
if this [] BindingObject then return false end if;
return some b [] this.localBindings satisfies b.qname = qname and b.enumerable
end proc;
proc Object_sealProperty(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Undefined
if phase = compile then
  throw a ConstantError exception — sealProperty cannot be called from a constant expression
end if;
if args > 1 then
  throw an ArgumentError exception — at most one argument can be supplied
end if;
arg: Object [] defaultArg(args, 0, true);
if arg = false then Evaluate sealObject(this) and ignore its result
elsif arg = true then
  Evaluate sealObject(this) and ignore its result;
  Evaluate sealAllLocalProperties(this) and ignore its result
elsif arg [] char16 [] String then
  throw a ReferenceError exception — property not found
end if;
gname: QualifiedName [] objectToQualifiedName(arg, phase);
Evaluate sealLocalProperty(this, gname) and ignore its result
end if;
return undefined
end proc;

17.2 Never


proc constructNever(c: Class, args: Object[], phase: Phase): Object
if args > 1 then
  throw an ArgumentError exception — at most one argument can be supplied
end if;
throw a TypeError exception — no coercions to Never are possible
end proc;

proc coerceNever(o: Object, c: Class): {none}
  return none
end proc;

17.3 Void

proc callVoid(this: Object, c: Class, args: Object[], phase: Phase): Undefined
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    return undefined
end proc;

proc constructVoid(c: Class, args: Object[], phase: Phase): Undefined
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    return undefined
end proc;

proc coerceVoid(o: Object, c: Class): {undefined, none}
    if o is null then return undefined else return none end if
end proc;

17.4 Null

Null: Class = new Class(localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
    complete: true, name: "Null", type: "object", dynamic: false, final: true, default: null,
    hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
    bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
    enumerate: ordinaryEnumerate, call: callNull, construct: constructNull, init: none, is: ordinaryIs,
    coerce: coerceNull[]

proc callNull(this: Object, c: Class, args: Object[], phase: Phase): Null
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    return null
end proc;

proc constructNull(c: Class, args: Object[], phase: Phase): Null
    note This function does not check phase and therefore can be used in a constant expression.
    if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
    end if;
    return null
end proc;

proc coerceNull(o: Object, c: Class): {null, none}
    if o is null then return o else return none end if
end proc;

17.5 Boolean

Boolean: Class = new Class(localBindings: {}, instanceProperties: {}, super: Object, prototype: BooleanPrototype,
    complete: true, name: "Boolean", type: "boolean", dynamic: false, final: true,
    default: false, hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead,
    bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
    write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: sameAsConstruct,
    construct: constructBoolean, init: none, is: ordinaryIs, coerce: coerceBoolean[]
proc constructBoolean(c: CLASS, args: OBJECT[], phase: PHASE): BOOLEAN
  note This function does not check phase and therefore can be used in a constant expression.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return objectToBoolean(defaultArg(args, 0, false))
end proc;

proc coerceBoolean(o: OBJECT, c: CLASS): BOOLEANOPT
  if o [] BOOLEAN then return o else return none end if
end proc;

BooleanPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
  stdConstBinding(public::"constructor", Class, Boolean),
  stdFunction(public::"toString", Boolean toString, 0),
  stdReserve(public::"valueOf", ObjectPrototype),
  archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
  env: none[]
}

proc Boolean toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in a constant expression.
  note This function ignores any arguments passed to it in args.
  a: BOOLEAN [] objectToBoolean(this);
  return objectToString(a, phase)
end proc;

17.6 GeneralNumber

GeneralNumber: CLASS = new CLASS[localBindings: {}, instanceProperties: {}, super: Object,
  prototype: GeneralNumberPrototype, complete: true, name: "GeneralNumber", typeofString: "object",
  dynamic: false, final: true, defaultValue: NaN, defaultHint: hintNumber,
  hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructGeneralNumber, init: none,
  is: ordinaryIs, coerce: coerceGeneralNumber[]

proc constructGeneralNumber(c: CLASS, args: OBJECT[], phase: PHASE): GENERALNUMBER
  note This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
  if |args| = 0 then return +zero
  elsif |args| = 1 then return objectToGeneralNumber(args[0], phase)
  else throw an ArgumentError exception — at most one argument can be supplied
  end if
end proc;

proc coerceGeneralNumber(o: OBJECT, c: CLASS): GENERALNUMBER [] {none}
  if o [] GENERALNUMBER then return o else return none end if
end proc;
**GeneralNumberPrototype**: `SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: { standardBinding: public: "constructor", Class, GeneralNumber),
standardFunction: public: "toString", GeneralNumber.prototype, toString, 1),
standardFunction: public: "toFixed", GeneralNumber.prototype, toFixed, 1),
standardFunction: public: "toExponential", GeneralNumber.prototype, toExponential, 1),
standardFunction: public: "toFixed", GeneralNumber.prototype, toFixed, 1),
```

**proc GeneralNumber.prototype.toString**: `((this: Object, f: SIMPLEINSTANCE, args: Object[], phase: Phase): String` 

**note** This function can be used in a constant expression if `this` and the argument can be converted to primitives in constant expressions.

**x**: `GeneralNumber{[]} objectToGeneralNumber(this, phase)`;
```

**proc GeneralNumber.prototype.toFixed**: `((this: Object, f: SIMPLEINSTANCE, args: Object[], phase: Phase): String` 

**note** This function can be used in a constant expression if `this` and the argument can be converted to primitives in constant expressions.

**x**: `GeneralNumber{[]} objectToGeneralNumber(this, phase)`;
```

**precisionLimit**: `INTEGER = an implementation-defined integer not less than 20;`
**proc** GeneralNumber_toExponential**(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING**

**note** This function can be used in a constant expression if **this** and the argument can be converted to primitives in constant expressions.

**note** This function is generic and can be applied even if **this** is not a general number.

if **args** > 1 then
  throw an ArgumentError exception — at most one argument can be supplied
end if;

**x: GENERALNUMBER** \[ objectToGeneralNumber**(this, phase)**; **fractionDigits: EXTENDEDINTEGER** \[ objectToExtendedInteger**(defaultArg(args, 0, NaN_f64), phase)**; if **fractionDigits** \[ \{+\infty, -\infty\} or
  (**fractionDigits** \[ NaN and (**fractionDigits** < 0 or **fractionDigits** > precisionLimit)) then
    throw a RangeError exception
end if;

if **x** [ FINTGENERALNUMBER] then return generalNumberToString**(x)** end if;

**r: RATIONAL** \[ toRational**(x)**;

**sign: STRING** \[ “-“;

if **r** < 0 then **sign** \[ “-“; **r** \[ -r end if;

**digits: STRING**;

**e: INTEGER**;

if **fractionDigits** \[ NaN then
  if **r** = 0 then **digits** \[ repeat**(‘0’, **fractionDigits** + 1); **e** \[ 0
else
  **e** \[ \log_10**(r)\]
  **n: INTEGER** \[ \text{floor}**(10^{**fractionDigits**-**e**} + 1/2)\]
  **note** At this point \( 10^{**fractionDigits**} \leq n \leq 10^{**fractionDigits**+1} \)
  if **n** \[ \text{floor}**(10^{**fractionDigits**+1}) then **n** \[ **n**/10; **e** \[ **e** + 1 end if;

  **digits** \[ integerToString**(n)**
end if;

  **note** At this point the string **digits** has exactly **fractionDigits** + 1 digits
else **r** \[ 0 then **digits** \[ “0”; **e** \[ 0
else **x** [ LONG [ ULONG then
  **digits** \[ integerToString**(r)**;
  **e** \[ |**digits**| - 1;
  while **digits**(\[|**digits**| - 1) = ‘0’ do **digits** \[ **digits**[0 ... |**digits**| - 2]
end while
else
  **k: INTEGER**;
  **s: INTEGER**;

  **case** **x** of
    **NONZEROFINITEFLOAT32** do
      Let **e**, **k**, and **s** be integers such that \( k \geq 1, 10^{k-1} \leq s \leq 10^k, (s\cdot10^{**e**+1-k})_{32} = x, **k** is as small as possible.

    **NONZEROFINITEFLOAT64** do
      Let **e**, **k**, and **s** be integers such that \( k \geq 1, 10^{k-1} \leq s \leq 10^k, (s\cdot10^{**e**+1-k})_{64} = x, **k** is as small as possible.
end case;

  **note** **k** is the number of digits in the decimal representation of **s**, **s** is not divisible by 10, and the least significant digit of **s** is not necessarily uniquely determined by the above criteria.

  When there are multiple possibilities for **s** according to the rules above, implementations are encouraged but not required to select the one according to the following rules: Select the value of **s** for which \( s\cdot10^{**e**+1-k} \) is closest in value to **r**; if there are two such possible values of **s**, choose the one that is even.

  **digits** \[ integerToString**(s)**
end if;

  return **sign** \[ exponentialNotationString**(digits, e)**
end proc;
proc GeneralNumber_toPrecision(this: Object, f: SimpleInstance, args: Object[], phase: Phase): String

**note** This function can be used in a constant expression if this and the argument can be converted to primitives in constant expressions.

**note** This function is generic and can be applied even if this is not a general number.

if args > 1 then
  throw an ArgumentError exception — at most one argument can be supplied
end if;

x: GeneralNumber[] objectToGeneralNumber(this, phase);
precision: ExtendedInteger[] objectToExtendedInteger(defaultArg, 0, NaN, phase);
if precision = NaN then return generalNumberToString(x) end if;
if precision ∈ {+∞, –∞} or precision < 1 or precision > precisionLimit + 1 then
  throw a RangeError exception
end if;

if this FiniteGeneralNumber then return generalNumberToString(x) end if;

r: Rational[] toRational(x);
sign: String[] “”;
if r < 0 then sign “-”; r –r end if;
digits: String;
e: Integer;
if r = 0 then digits repeat(“0”, precision); e 0
else
  e ⌊log_{10}(r)⌋
  n: Integer[] ⌊10^{precision-1}*e + 1/2⌋
  **note** At this point 10^{precision-1} ≤ n ≤ 10^{precision}
  if n = 10^{precision} then n n/10; e e + 1 end if;
digits integerToString(n)
end if;

**note** At this point the string digits has exactly precision digits

if e < –6 or e ≥ precision then return sign exponentialNotationString(digits, e)
else if e = precision - 1 then return sign digits
else if e ≥ 0 then return sign digits[0 ... e] “,” digits[e + 1 ...]
else return sign “0 .” repeat(“0”, -(e + 1)) digits
end if

end proc;

17.7 long

long: Class = new Class[
  localBindings: {
    stdConstBinding(public:”MAX_VALUE”, ulong, (2^{63} - 1)_{long}),
    stdConstBinding(public:”MIN_VALUE”, ulong, (-2^{63})_{long}),
  }, super: GeneralNumber, prototype: longPrototype, complete: true, name: “long”,
typeOfString: “long”, dynamic: false, final: true, defaultValue: 0_{long}, hasProperty: ordinaryHasProperty,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructLong, init: none, is: ordinaryIs,
coerce: coerceLong]
proc constructLong(c: CLASS, args: OBJECT[], phase: PHASE): LONG
  note This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT[] defaultArg(args, 0, +zero_{64});
  i: INTEGER[] objectToInteger(arg, phase);
  if –2^{63} ≤ i ≤ 2^{63} – 1 then return i_{long}
  else throw a RangeError exception — i is out of the LONG range
  end if;
end proc;

proc coerceLong(o: OBJECT, c: CLASS): LONG[] {none}
  if o [] GENERALNUMBER then return none end if;
  i: INTEGEROPT[] checkInteger(o);
  if i ≠ none and –2^{63} ≤ i ≤ 2^{63} – 1 then return i_{long}
  else throw a RangeError exception — i is out of the LONG range
  end if;
end proc;

longPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
  stdConstBinding(public::"constructor", Class, long),
  stdReserve(public::"toString", GeneralNumberPrototype),
  stdReserve(public::"valueOf", GeneralNumberPrototype),
  archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
  construct: none, env: none}]

17.8 ulong

ulong: CLASS = new CLASS[localBindings: {
  stdConstBinding(public::"MAX_VALUE", ulong, (2^{64} – 1)_{ulong}),
  stdConstBinding(public::"MIN_VALUE", ulong, 0_{ulong}),
  instanceProperties: {}, super: GeneralNumber, prototype: ulongPrototype, complete: true, name: "ulong",
  toString: "ulong", dynamic: false, final: true, defaultValue: 0_{ulong}, hasProperty: ordinaryHasProperty,
  bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructULong, init: none, is: ordinaryIs,
  coerce: coerceULong[]
}

proc constructULong(c: CLASS, args: OBJECT[], phase: PHASE): ULONG
  note This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT[] defaultArg(args, 0, +zero_{64});
  i: INTEGER[] objectToInteger(arg, phase);
  if 0 ≤ i ≤ 2^{64} – 1 then return i_{ulong}
  else throw a RangeError exception — i is out of the ULONG range
  end if;
end proc;
proc coerceULong(o: Object, c: Class): ULONG [] {none}
if o [] GENERALNUMBER then return none end if;
i: INTEGEROPT [] checkInteger(o);
if i # none and 0 ≤ i ≤ 64 – 1 then return i_ulong
else throw a RangeError exception — i is out of the ULONG range end if
end proc;

ulongPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
    stdConstBinding(public:“constructor”, Class, ulong),
    stdReserve(public:“toString”, GeneralNumberPrototype),
    stdReserve(public:“valueOf”, GeneralNumberPrototype),
    archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
    construct: none, env: none[]
}

17.9 float

float: CLASS = new CLASS[localBindings: {
    stdConstBinding(public:“MAX_VALUE”, float, (3.4028235×10^{38})_{1022}),
    stdConstBinding(public:“MIN_VALUE”, float, (10^{-38})_{1022}),
    stdConstBinding(public:“NaN”, float, NaN_{1022}),
    stdConstBinding(public:“NEGATIVE_INFINITY”, float, -∞_{1022}),
    stdConstBinding(public:“POSITIVE_INFINITY”, float, +∞_{1022}),
    instanceProperties: {}, super: GeneralNumber, prototype: floatPrototype, complete: true, name: “float”,
typeOfString: “float”, dynamic: false, final: true, defaultValue: NaN_{1022}, hasProperty: ordinaryHasProperty,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructFloat, init: none, is: ordinaryIs,
    coerce: coerceFloat[]
}

proc constructFloat(c: Class, args: Object[], phase: Phase): FLOAT32
    note This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
    if |args| = 0 then return +ZERO_{1022}
    elsif |args| = 1 then return objectToFloat32(args[0], phase)
    else throw an ArgumentError exception — at most one argument can be supplied end if
end proc;

proc coerceFloat(o: Object, c: Class): FLOAT32 [] {none}
if o [] GENERALNUMBER then return toFloat32(o) else return none end if
end proc;

floatPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
    stdConstBinding(public:“constructor”, Class, float),
    stdReserve(public:“toString”, GeneralNumberPrototype),
    stdReserve(public:“valueOf”, GeneralNumberPrototype),
    archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
    construct: none, env: none[]
}
17.10 Number

Number: CLASS = new CLASS[]localBindings: {
  stdConstBinding(public: "MAX_VALUE", Number, 1.7976931348623157e308),
  stdConstBinding(public: "MIN_VALUE", Number, -5e324),
  stdConstBinding(public: "NaN", Number, NaN),
  stdConstBinding(public: "NEGATIVE_INFINITY", Number, -∞),
  stdConstBinding(public: "POSITIVE_INFINITY", Number, +∞),
}

instanceProperties: {}, super: GeneralNumber, prototype: NumberPrototype, complete: true,
name: "Number", typeofString: "number", dynamic: false, final: true, defaultValue: NaN,
hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructNumber, init: none, is: ordinaryIs,
coerce: coerceNumber

proc constructNumber(c: CLASS, args: OBJECT[], phase: PHASE): FLOAT64

  note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant
        expression.

  if |args| = 0 then return +zero
  elsif |args| = 1 then return objectToFloat64(args[0], phase)
  else throw an ArgumentError exception — at most one argument can be supplied
end if

end proc;

proc coerceNumber(o: OBJECT, c: CLASS): FLOAT64 [] {none}

  if o [] GENERALNUMBER then return toFloat64(o) else return none end if
end proc;

NumberPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[]localBindings: {
  stdConstBinding(public: "constructor", Class, Number),
  stdReserve(public: "toString", GeneralNumberPrototype),
  stdReserve(public: "valueOf", GeneralNumberPrototype),
}

archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
construct: none, env: none[]
proc makeBuiltInIntegerClass(name: STRING, low: INTEGER, high: INTEGER): CLASS
proc construct(c: CLASS, args: OBJECT[], phase: PHASE): FLOAT64

   note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
if [args] > 1 then
   throw an ArgumentError exception — at most one argument can be supplied
end if;
arg: OBJECT [] defaultArg(args, 0, +zero_64);
x: FLOAT64 objectToFloat64(arg, phase);
i: INTEGEROPT [] checkInteger(x);
if i ≠ none and low ≤ i ≤ high then
   note –zero_64 is coerced to +zero_64.
   return i_64
end if;
throw a RangeError exception
end proc;
proc is(o: OBJECT, c: CLASS): BOOLEAN
if o FLOAT64 then return false end if;
i: INTEGEROPT [] checkInteger(o);
return i ≠ none and low ≤ i ≤ high
end proc;
proc coerce(o: OBJECT, c: CLASS): FLOAT64 [] {none}
if o GENERALNUMBER then return none end if;
i: INTEGEROPT [] checkInteger(o);
if i ≠ none and low ≤ i ≤ high then
   note –zero_32, +zero_32, and –zero_64 are all coerced to +zero_64.
   return i_64
end if;
throw a RangeError exception
end proc;
return new CLASS[]localBindings: {
   stdConstBinding(public::"MAX_VALUE", Number, high_64),
   stdConstBinding(public::"MIN_VALUE", Number, low_64),
   instanceProperties: {}, super: Number, prototype: Number.prototype, complete: true, name: name,
   typeofString: "number", dynamic: false, final: true, defaultValue: +zero_64,
   hasProperty: Number.hasProperty, bracketRead: Number.bracketRead,
   bracketWrite: Number.bracketWrite, bracketDelete: Number.bracketDelete, read: Number.read,
   write: Number.write, delete: Number.delete, enumerate: Number.enumerate, call: sameAsConstruct,
   construct: construct, init: none, is: is, coerce: coerce[]
end proc;
sbyte: CLASS = makeBuiltInIntegerClass("sbyte", –128, 127);
byte: CLASS = makeBuiltInIntegerClass("byte", 0, 255);
short: CLASS = makeBuiltInIntegerClass("short", –32768, 32767);
ushort: CLASS = makeBuiltInIntegerClass("ushort", 0, 65535);
int: CLASS = makeBuiltInIntegerClass("int", –2147483648, 2147483647);
uint: CLASS = makeBuiltInIntegerClass("uint", 0, 4294967295);
17.11 char

char: CLASS = new CLASS[]localBindings: {stdFunction(public::“fromCharCode”, char_fromCharCode, 1)},
    instanceProperties: {}, super: Object, prototype: charPrototype, complete: true, name: “char”,
typeOfString: “char”, dynamic: false, final: true, defaultValue: ‘«NUL»’, hasProperty: ordinaryHasProperty,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, 
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructChar, init: none, is: ordinaryIs,
    coerce: coerceChar]

proc callChar(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): CHAR16
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    s: STRING [] objectToString(args[0], phase);
    if |s| ≠ 1 then throw a RangeError exception — only one character may be given end if;
    return s[0]
end proc;

proc constructChar(c: CLASS, args: OBJECT[], phase: PHASE): CHAR16
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
    if |args| > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT [] defaultArg(args, 0, undefined);
    if arg = undefined then return ‘«NUL»’
    elsif arg [] CHAR16 then return arg
    else
        s: STRING [] objectToString(args[0], phase);
        if |s| ≠ 1 then throw a RangeError exception — only one character may be given
        end if;
        return s[0]
    end if
end proc;

proc coerceChar(o: OBJECT, c: CLASS): CHAR16 [] {none}
    if o [] CHAR16 then return o else return none end if
end proc;

proc char_fromCharCode(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
    note  This function can be used in a constant expression if the argument can be converted to a primitive in a constant expression.
    if |args| ≠ 1 then
        throw an ArgumentError exception — exactly one argument must be supplied
    end if;
    i: INTEGER [] objectToInteger(args[0], phase);
    if 0 ≤ i ≤ 0xFFFF then return integerToChar16(i)
    else throw a RangeError exception — character code out of range
    end if
end proc;
charPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE\[localBindings: {
    stdConstBinding(public: “constructor”, Class, char),
    stdReserve(public: “toString”, StringPrototype),
    stdReserve(public: “valueOf”, StringPrototype),
    archetype: StringPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
    env: none\]

17.12 String

String: CLASS = new CLASS\[localBindings: {stdFunction(public: “fromCharCode”, String_fromCharCode, 1)},
    instanceProperties: {
    new INSTANCEGETTER\[multiname: {public: “length”}, final: true, enumerable: false, call: String_length\],
    dynamic: false, final: true, defaultValue: null, hasProperty: stringHasProperty,
    bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
    bracketDelete: ordinaryBracketDelete, read: readString, write: ordinaryWrite, delete: ordinaryDelete,
    enumerate: ordinaryEnumerate, call: sameAsConstruct, construct: constructString, init: none, is: ordinaryIs,
    coerce: coerceString\]

proc stringHasProperty(o: Object, c: Class, property: Object, flat: Boolean, phase: Phase): Boolean
    note o □ String because stringHasProperty is only called on instances of class String.
    qname: QUALIFIEDNAME □ objectToQualifiedName(property, phase);
    i: INTEGEROPT ▲ multinameToUnsignedInteger(qname);
    if i ≠ none then return i < o
    else
        return findBaseInstanceProperty(c, {qname}, read) ≠ none or
        findBaseInstanceProperty(c, {qname}, write) ≠ none or
        findArchetypeProperty(o, {qname}, read, flat) ≠ none or
        findArchetypeProperty(o, {qname}, write, flat) ≠ none
    end if
end proc;

proc readString(o: Object, limit: Class, multiname: Multiname, env: ENVIRONMENTOPT,
    undefinedIfMissing: Boolean, phase: Phase): ObjectOPT
    note o □ String because readString is only called on instances of class String.
    if limit = String then
        i: INTEGEROPT ▲ multinameToUnsignedInteger(multiname);
        if i ≠ none then
            if i < o then return o[i]
            elsif i < o then return undefined
            else return undefinedIfMissing
        end if
    end if;
    return ordinaryRead(o, limit, multiname, env, undefinedIfMissing, phase)
end proc;

proc constructString(c: Class, args: Object[], phase: Phase): String
    note This function can be used in a constant expression if the argument can be converted to a primitive in a constant
    expression.
    if [args] = 0 then return “”
    elsif [args] = 1 then return objectToString(args[0], phase)
    else throw an ArgumentError exception — at most one argument can be supplied
end if
end proc;
proc coerceString(o: Object, c: Class): String = null {} none
if o = null = String then return o
elsif o = char16 then return [o]
else return none
end if
end proc;

proc String_length(this: Object, phase: Phase): Object
  note this = string because this getter cannot be extracted from the String class.
  length: Integer = [this];
  return length 64
end proc;

proc String_fromCharCode(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
  note This function can be used in a constant expression if the arguments can be converted to primitives in constant expressions.
s: String = "";
for each arg = args do
  i: Integer = objectToInteger(arg, phase);
  if 0 ≤ i ≤ 0x10FFFF then s = s ⊕ integerToUTF16(i)
  else throw a RangeError exception — character code out of range
end if
end for each;
return s
end proc;

StringPrototype: SimpleInstance = new SimpleInstanceLocalBindings: {
  stdConstBinding(public: "constructor", Class, String),
  stdFunction(public: "toString", String_toString, 0),
  stdReserve(public: "valueOf", ObjectPrototype),
  stdFunction(public: "charAt", String_charAt, 1),
  stdFunction(public: "charCodeAt", String_charCodeAt, 1),
  stdFunction(public: "concat", String_concat, 1),
  stdFunction(public: "indexToString", String_indexToString, 1),
  stdFunction(public: "lastIndexOf", String_lastIndexOf, 1),
  stdFunction(public: "localeCompare", String_localeCompare, 1),
  stdFunction(public: "match", String_match, 1),
  stdFunction(public: "replace", String_replace, 1),
  stdFunction(public: "search", String_search, 1),
  stdFunction(public: "slice", String_slice, 2),
  stdFunction(public: "split", String_split, 2),
  stdFunction(public: "substring", String_substring, 2),
  stdFunction(public: "toLowerCase", String_toLowerCase, 0),
  stdFunction(public: "toLocaleLowerCase", String_toLocaleLowerCase, 0),
  stdFunction(public: "toUpperCase", String_toUpperCase, 0),
  stdFunction(public: "toLocaleUpperCase", String_toLocaleUpperCase, 0),
}

proc String_toString(this: Object, f: SimpleInstance, args: Object[], phase: Phase): String
  note This function can be used in a constant expression if this can be converted to a primitive in a constant expression.
  note This function is generic and can be applied even if this is not a string.
  note This function ignores any arguments passed to it in args.
  return objectToString(this, phase)
end proc;
proc String_charAt(this: Object, f: SimpleInstance, args: Object[], phase: Phase): STRING
    note This function can be used in a constant expression if this and the argument can be converted to primitives in constant expressions.
    note This function is generic and can be applied even if this is not a string.
    if [args] > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    s: STRING[] objectToString(this, phase);
    position: ExtendedInteger[] objectToExtendedInteger(defaultArg(args, 0, +zero_64), phase);
    if position = NaN then throw a RangeError exception
    elsif position [+∞, -∞) and 0 ≤ position < |s| then return [s[position]]
    else return "" end if
end proc;

proc String_charCodeAt(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Float64
    note This function can be used in a constant expression if this and the argument can be converted to primitives in constant expressions.
    note This function is generic and can be applied even if this is not a string.
    if [args] > 1 then
        throw an ArgumentError exception — at most one argument can be supplied
    end if;
    s: STRING[] objectToString(this, phase);
    position: ExtendedInteger[] objectToExtendedInteger(defaultArg(args, 0, +zero_64), phase);
    if position = NaN then throw a RangeError exception
    elsif position [+∞, -∞) and 0 ≤ position < |s| then return (char16ToInteger(s[position]))_64
    else return NaN_64 end if
end proc;

proc String_concat(this: Object, f: SimpleInstance, args: Object[], phase: Phase): STRING
    note This function can be used in a constant expression if this and the argument can be converted to primitives in constant expressions.
    note This function is generic and can be applied even if this is not a string.
    s: STRING[] objectToString(this, phase);
    for each arg[] args do s [] s + objectToString(arg, phase) end for each;
    return s end proc;

proc String_indexOf(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Float64
    note This function can be used in a constant expression if this and the arguments can be converted to primitives in constant expressions.
    note This function is generic and can be applied even if this is not a string.
    if [args] [1, 2] then
        throw an ArgumentError exception — at least one and at most two arguments must be supplied
    end if;
    s: STRING[] objectToString(this, phase);
    pattern: STRING[] objectToString(args[0], phase);
    arg: Object[] defaultArg(args, 1, +zero_64);
    position: Integer[] pinExtendedInteger(objectToExtendedInteger(arg, phase), |s|, false);
    while position + |pattern| ≤ |s| do
        if s[position ... position + |pattern| - 1] = pattern then return position_64
    end if;
    position [] position + 1
end while;
return (-1)_64 end proc;
proc String_lastIndexOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64

  note This function can be used in a constant expression if this and the arguments can be converted to primitives in constant expressions.

  note This function is generic and can be applied even if this is not a string.

if [args] [] {1, 2} then
  throw an ArgumentError exception — at least one and at most two arguments must be supplied
end if;

s: STRING [] objectToString(this, phase);
pattern: STRING [] objectToString(args[0], phase);
arg: OBJECT [] defaultArg(args, 1, 4∞64);
position: INTEGER [] pinExtendedInteger(objectToExtendedInteger(arg, phase), |s|, false);
if position + |pattern| > |s| then position [] |s| – |pattern| end if;
while position ≥ 0 do
  if s[position ... position + |pattern| – 1] = pattern then return position64
end if;
  position [] position – 1
end while;
return (–1)64
end proc;

proc String_localeCompare(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): FLOAT64

  note This function is generic and can be applied even if this is not a string.

if phase = compile then
  throw a ConstantError exception — localeCompare cannot be called from a constant expression
end if;
if |args| ≠ 1 then
  throw an ArgumentError exception — at least one argument must be supplied
end if;

s1: STRING [] objectToString(this, phase);
s2: STRING [] objectToString(args[0], phase);

Let result: OBJECT be a value of type Number that is the result of a locale-sensitive string comparison of s1 and s2. The two strings are compared in an implementation-defined fashion. The result is intended to order strings in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether s1 comes before s2 in the sort order, they are equal, or s1 comes after s2 in the sort order, respectively. The result shall not be NaN64. The comparison shall be a consistent comparison function on the set of all strings.

return result
end proc;

proc String_match(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT

  note This function is generic and can be applied even if this is not a string.

if phase = compile then
  throw a ConstantError exception — match cannot be called from a constant expression
end if;
if |args| ≠ 1 then
  throw an ArgumentError exception — exactly one argument must be supplied
end if;
s: STRING [] objectToString(this, phase);

Evaluate ???? and ignore its result
end proc;
proc String_replace(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function is generic and can be applied even if this is not a string.
if phase = compile then
    throw a ConstantError exception — replace cannot be called from a constant expression
end if;
if |args| ≠ 2 then
    throw an ArgumentError exception — exactly two arguments must be supplied
end if;
s: STRING [] objectToString(this, phase);
Evaluate ???? and ignore its result
end proc;

proc String_search(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function is generic and can be applied even if this is not a string.
if phase = compile then
    throw a ConstantError exception — search cannot be called from a constant expression
end if;
if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
end if;
s: STRING [] objectToString(this, phase);
Evaluate ???? and ignore its result
end proc;

proc String_slice(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
note This function can be used in a constant expression if this and the arguments can be converted to primitives in constant expressions.
note This function is generic and can be applied even if this is not a string.
if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
end if;
s: STRING [] objectToString(this, phase);
startArg: OBJECT [] defaultArg(args, 0, +zero64);
endArg: OBJECT [] defaultArg(args, 1, +inf64);
start: INTEGER [] pinExtendedInteger(objectToExtendedInteger(startArg, phase), |s|, true);
end: INTEGER [] pinExtendedInteger(objectToExtendedInteger(endArg, phase), |s|, true);
if start < end then return s[start ... end - 1] else return "" end if
end proc;

proc String_split(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function is generic and can be applied even if this is not a string.
if phase = compile then
    throw a ConstantError exception — split cannot be called from a constant expression
end if;
if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
end if;
s: STRING [] objectToString(this, phase);
Evaluate ???? and ignore its result
end proc;
**String_substring**

**Note**
This function can be used in a constant expression if `this` and the arguments can be converted to primitives in constant expressions.

```plaintext
if [args] > 2 then
  throw an ArgumentError exception — at most two arguments can be supplied
end if;

s: STRING [] objectToString(this, phase);
startArg: OBJECT [] defaultArg(args, 0, +zero_{164});
endArg: OBJECT [] defaultArg(args, 1, +∞_{164});
start: INTEGER [] pinExtendedInteger(objectToExtendedInteger(startArg, phase), |s|, false);
end: INTEGER [] pinExtendedInteger(objectToExtendedInteger(endArg, phase), |s|, false);
if start ≤ end then return s[start ... end – 1]
else return s[end ... start – 1]
end if
end proc;
```

**String_toLowerCase**

**Note**
This function can be used in a constant expression if `this` can be converted to a primitive in a constant expression.

```plaintext
s: STRING [] objectToString(this, phase);
s32: CHAR21 [] stringToUTF32(s);
r: STRING [] “”;
for each ch s32 do r r charToLowerFull(ch) end for each;
return r
end proc;
```

**String_toLocaleLowerCase**

**Note**
This function is generic and can be applied even if `this` is not a string.

```plaintext
if phase = compile then
  throw a ConstantError exception — toLocaleLowerCase cannot be called from a constant expression
end if;

s: STRING [] objectToString(this, phase);
s32: CHAR21 [] stringToUTF32(s);
r: STRING [] “”;
for each ch s32 do r r charToLowerLocalized(ch) end for each;
return r
end proc;
```

**String_toUpperCase**

**Note**
This function can be used in a constant expression if `this` can be converted to a primitive in a constant expression.

```plaintext
s: STRING [] objectToString(this, phase);
s32: CHAR21 [] stringToUTF32(s);
r: STRING [] “”;
for each ch s32 do r r charToUpperFull(ch) end for each;
return r
end proc;
```
null
writeArrayPrivateLength(array, length, phase) sets an Array’s private length to length after ensuring that length is between 0 and arrayLimit inclusive. See also writeLength, which can work on non-Array objects.

```
proc writeArrayPrivateLength(array: OBJECT, length: INTEGER, phase: {run})
  if length < 0 or length > arrayLimit then
    throw a RangeError exception — array length out of range
  end if;
  Evaluate dotWrite(array, {arrayPrivate::"length"}, lengthf64, phase) and ignore its result
end proc;
```

```
proc multinameToUnsignedInteger(multiname: MULTINAME): INTEGEROPT
  if |multiname| ≠ 1 then return none end if;
  qname: QNAME = the one element of multiname;
  if qname.namespace ≠ public then return none end if;
  name: STRING = qname.id;
  if name ≠ [] then
    if name = "0" then return 0
    elsif name[0] ≠ '0' and (every ch in name satisfies ch satisfies {'0' ... '9'}) then
      return stringToExtendedInteger(name)
    end if
  end if;
  return none
end proc;
```

```
proc initArray(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
  if |args| = 1 then
    arg: OBJECT = args[0];
    if arg GENERALNUMBER then
      length: INTEGEROPT = checkInteger(arg);
      if length = none then
        throw a RangeError exception — array length must be an integer
      end if;
      Evaluate writeArrayPrivateLength(this, length, phase) and ignore its result;
      return
    end if;
    i: INTEGER = 0;
    for each arg in args do
      Evaluate indexWrite(this, i, arg, phase) and ignore its result;
      i = i + 1
    end for each;
    note The call to indexWrite above also set the array’s length to i.
  end if;
end proc;
```

```
proc Array_getLength(this: OBJECT, phase: PHASE): FLOAT64
  note is(this, Array) because this getter cannot be extracted from the Array class.
  note An array’s length is mutable, so reading it will throw ConstantError when phase = compile.
  return readInstanceSlot(this, arrayPrivate::"length", phase)
end proc;
```
proc Array_setLength(this: OBJECT, length: OBJECT, phase: PHASE)
note  is(this, Array) because this setter cannot be extracted from the Array class.
if phase = compile then
  throw a ConstantError exception — an array’s length cannot be set from a constant expression
end if;
newLength: INTEGEROPT [] checkInteger(objectToGeneralNumber(length, phase));
if newLength = none or newLength < 0 or newLength > arrayLimit then
  throw a RangeError exception — array length out of range or not an integer
end if;
oldLength: INTEGER [] readArrayPrivateLength(this, phase);
if newLength < oldLength then
  note Delete all indexed properties greater than or equal to the new length
  proc qnameInDeletedRange(qname: QUALIFIEDNAME): BOOLEAN
  i: INTEGEROPT [] multinameToUnsignedInteger({qname});
  return i ≠ none and newLength ≤ i < oldLength
  end proc;
  this.localBindings [] { b | [] b [] this.localBindings such that not qnameInDeletedRange(b.qname)}
end if;
Evaluate writeArrayPrivateLength(this, newLength, phase) and ignore its result
end proc;

ArrayPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
  stdConstBinding(public::"constructor", Class, Array),
  stdFunction(public::"toString", Array_toString, 0),
  stdFunction(public::"toLocaleString", Array_toLocaleString, 0),
  stdFunction(public::"concat", Array_concat, 1),
  stdFunction(public::"join", Array_join, 1),
  stdFunction(public::"pop", Array_pop, 0),
  stdFunction(public::"push", Array_push, 1),
  stdFunction(public::"reverse", Array_reverse, 0),
  stdFunction(public::"shift", Array_shift, 0),
  stdFunction(public::"slice", Array_slice, 2),
  stdFunction(public::"sort", Array_sort, 1),
  stdFunction(public::"splice", Array_splice, 2),
  stdFunction(public::"unshift", Array_unshift, 1)},

proc Array_toString(this: OBJECT, f: SIMPLEINSTANCE, args OBJECT[], phase: PHASE): STRING
if phase = compile then
  throw a ConstantError exception — toString cannot be called on an Array from a constant expression
end if;
note  This function is generic and can be applied even if this is not an Array.
note  This function ignores any arguments passed to it in args.
return internalJoin(this, ",", phase)
end proc;
proc Array_toLocaleString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
if phase = compile then
    throw a ConstantError exception — toLocaleString cannot be called on an Array from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.

note This function passes any arguments passed to it in args to toLocaleString applied to the elements of the array.

separator: STRING[] the list-separator string appropriate for the host’s current locale, derived in an implementation-defined way;
length: INTEGER[] readLength(this, phase);
result: STRING[] ""

i: INTEGER[] 0;
while i ≠ length do
    elt: OBJECTOPT[] indexRead(this, i, phase);
    if elt = {undefined, null, none} then
        toLocaleStringMethod: OBJECT[] dotRead(elt, {public::"toLocaleString"}, phase);
s: OBJECT[] call(elt, toLocaleStringMethod, args, phase);
        if s ◄ CHAR16 ◄ STRING then
            throw a TypeError exception — toLocaleString should return a string
        end if;
        result ◄ result ◄ toString(s)
    end if;
    i ◄ i + 1;
    if i ≠ length then result ◄ result ◄ separator end if
end while;
return result
end proc;

proc Array_concat(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
if phase = compile then
    throw a ConstantError exception — concat cannot be called from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.

constituents: OBJECT[] [this] ◄ args;
array: OBJECT[] construct(Array, [], phase);
i: INTEGER[] 0;
for each o ◄ constituents do
    if is(o, Array) then
        oLength: INTEGER[] readLength(o, phase);
k: INTEGER[] 0;
    while k ≠ oLength do
        elt: OBJECTOPT[] indexRead(o, k, phase);
        if elt ≠ none then
            Evaluate indexWrite(array, i, elt, phase) and ignore its result
        end if;
        k ◄ k + 1;
i ◄ i + 1
    end while
    else Evaluate indexWrite(array, i, o, phase) and ignore its result; i ◄ i + 1
end if
end for each;
Evaluate writeArrayPrivateLength(array, i, phase) and ignore its result;
return array
end proc;
proc \texttt{Array\_join(this: Object, f: Simple\_Instance, args: Object[ ], phase: Phase): String}

if \texttt{phase} = \texttt{compile} then
    throw a \texttt{ConstantError} exception — \texttt{join} cannot be called from a constant expression
end if;

\textbf{note} This function is generic and can be applied even if \texttt{this} is not an \texttt{Array}.

if \textbf{len} \texttt{args} > 1 then
    throw an \texttt{ArgumentError} exception — at most one argument can be supplied
end if;

\texttt{arg: Object \texttt{defaultArg(args, 0, undefined)}};

\texttt{separator: String \texttt{"", \texttt{""}}};

if \texttt{arg} ≠ \texttt{undefined} then \texttt{separator \texttt{objectToString(arg, phase)}} end if;

\texttt{return internalJoin(this, separator, phase)}
end proc;

proc \texttt{internalJoin(this: Object, separator: String, phase: \{run\}): String}

\texttt{length: Integer \texttt{readLength(this, phase)}};

\texttt{result: String \texttt{""}};

\texttt{i: Integer \texttt{0}};

while \texttt{i} ≠ \texttt{length} do
    \texttt{elt: Object\texttt{OPT indexRead(this, i, phase)}};
    if \texttt{elt} \texttt{\{undefined, null, none\}} then
        \texttt{result \texttt{result \texttt{objectToString(elt, phase)}}}
    end if;
    \texttt{i \texttt{i + 1}};
    if \texttt{i} ≠ \texttt{length} then \texttt{result \texttt{result \texttt{separator}}} end if
end while;

\texttt{return result}
end proc;

proc \texttt{Array\_pop(this: Object, f: Simple\_Instance, args: Object[ ], phase: Phase): Object}

if \texttt{phase} = \texttt{compile} then
    throw a \texttt{ConstantError} exception — \texttt{pop} cannot be called from a constant expression
end if;

\textbf{note} This function is generic and can be applied even if \texttt{this} is not an \texttt{Array}.

if \textbf{len} \texttt{args} ≠ 0 then throw an \texttt{ArgumentError} exception — no arguments can be supplied
end if;

\texttt{length: Integer \texttt{readLength(this, phase)}};

\texttt{result: Object \texttt{undefined}};

if \texttt{length} ≠ 0 then
    \texttt{length \texttt{length \texttt{1}}};
    \texttt{elt: Object\texttt{OPT indexRead(this, length, phase)}};
    if \texttt{elt} \texttt{none} then
        \texttt{result \texttt{elt}};
        Evaluate \texttt{indexWrite(this, length, none, phase)} and ignore its result
    end if
end if;

Evaluate \texttt{writeLength(this, length, phase)} and ignore its result;

\texttt{return result}
end proc;
proc $\text{Array\_push}(this: \text{OBJECT}, f: \text{SIMPLE\_INSTANCE}, args: \text{OBJECT}[], phase: \text{PHASE}): \text{OBJECT}$

if phase = compile then
  throw a $\text{ConstantError}$ exception — push cannot be called from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.

length: INTEGER $\equiv$ readLength(this, phase);

for each arg in args do
  Evaluate indexWrite(this, length, arg, phase) and ignore its result;
  length $\equiv$ length + 1
end for each;

Evaluate writeLength(this, length, phase) and ignore its result;

return length
end proc;

proc $\text{Array\_reverse}(this: \text{OBJECT}, f: \text{SIMPLE\_INSTANCE}, args: \text{OBJECT}[], phase: \text{PHASE}): \text{OBJECT}$

if phase = compile then
  throw a $\text{ConstantError}$ exception — reverse cannot be called from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.

if $|args| \neq 0$ then throw an $\text{ArgumentError}$ exception — no arguments can be supplied
end if;

length: INTEGER $\equiv$ readLength(this, phase);

lo: INTEGER $\equiv$ 0;
hi: INTEGER $\equiv$ length – 1;

while lo < hi do
  loEl: OBJECTOPT $\equiv$ indexRead(this, lo, phase);
  hiEl: OBJECTOPT $\equiv$ indexRead(this, hi, phase);
  Evaluate indexWrite(this, lo, hiEl, phase) and ignore its result;
  Evaluate indexWrite(this, hi, loEl, phase) and ignore its result;
  lo $\equiv$ lo + 1;
  hi $\equiv$ hi – 1
end while;

return this
end proc;
proc $\text{Array\_shift}(\text{this}: \text{OBJECT}, f: \text{SIMPLE\_INSTANCE}, \text{args}: \text{OBJECT}[], \text{phase}: \text{PHASE}): \text{OBJECT}$

if $\text{phase} = \text{compile}$ then
    throw a $\text{ConstantError}$ exception — $\text{shift}$ cannot be called from a constant expression
end if;

note This function is generic and can be applied even if $\text{this}$ is not an $\text{Array}$.
if $|\text{args}| \neq 0$ then throw an $\text{ArgumentError}$ exception — no arguments can be supplied
end if;

$\text{length}: \text{INTEGER} \; \text{readLength}(\text{this}, \text{phase})$;
$\text{result}: \text{OBJECT} \; \text{undefined}$;

if $\text{length} \neq 0$ then
    $\text{elt}: \text{OBJECT\_OPT} \; \text{indexRead}(\text{this}, 0, \text{phase})$;
    if $\text{elt} \neq \text{none}$ then result $\; \text{elt}$ end if;
    $\text{i}: \text{INTEGER} \; 1$;
    while $\text{i} \neq \text{length}$ do
        $\text{elt} \; \text{indexRead}(\text{this}, \text{i}, \text{phase})$;
        Evaluate $\text{indexWrite}(\text{this}, \text{i} - 1, \text{elt}, \text{phase})$ and ignore its result;
        $\text{i} \; \text{i} + 1$
    end while;
    $\text{length} \; \text{length} - 1$
    Evaluate $\text{indexWrite}(\text{this}, \text{length}, \text{none}, \text{phase})$ and ignore its result
end if;
Evaluate $\text{writeLength}(\text{this}, \text{length}, \text{phase})$ and ignore its result;
return result
end proc;

proc $\text{Array\_slice}(\text{this}: \text{OBJECT}, f: \text{SIMPLE\_INSTANCE}, \text{args}: \text{OBJECT}[], \text{phase}: \text{PHASE}): \text{OBJECT}$

if $\text{phase} = \text{compile}$ then
    throw a $\text{ConstantError}$ exception — $\text{slice}$ cannot be called on an $\text{Array}$ from a constant expression
end if;

note This function is generic and can be applied even if $\text{this}$ is not an $\text{Array}$.
if $|\text{args}| > 2$ then
    throw an $\text{ArgumentError}$ exception — at most two arguments can be supplied
end if;

$\text{length}: \text{INTEGER} \; \text{readLength}(\text{this}, \text{phase})$;
$\text{startArg}: \text{OBJECT} \; \text{defaultArg}(\text{args}, 0, +\text{zero}_{16})$;
$\text{endArg}: \text{OBJECT} \; \text{defaultArg}(\text{args}, 1, +\infty_{16})$;
$\text{start}: \text{INTEGER} \; \text{pinExtendedInteger}(\text{objectToExtendedInteger}(\text{startArg}, \text{phase}), \text{length}, \text{true})$;
$\text{end}: \text{INTEGER} \; \text{pinExtendedInteger}(\text{objectToExtendedInteger}(\text{endArg}, \text{phase}), \text{length}, \text{true})$;
return $\text{makeArraySlice}(\text{this}, \text{start}, \text{end}, \text{phase})$
end proc;

proc $\text{makeArraySlice}(\text{array}: \text{OBJECT}, \text{start}: \text{INTEGER}, \text{end}: \text{INTEGER}, \text{phase}: \{\text{run}\}): \text{OBJECT}$
slice: $\text{OBJECT} \; \text{construct}(\text{array}, \{\}, \text{phase})$
$i: \text{INTEGER} \; \text{start}$;
$j: \text{INTEGER} \; 0$
while $i < \text{end}$ do
    $\text{elt}: \text{OBJECT\_OPT} \; \text{indexRead}(\text{array}, i, \text{phase})$;
    Evaluate $\text{indexWrite}(\text{slicer}, j, \text{elt}, \text{phase})$ and ignore its result;
    $\text{i} \; \text{i} + 1$
    $\text{j} \; \text{j} + 1$
end while;
Evaluate $\text{writeLength}(\text{slicer}, j, \text{phase})$ and ignore its result;
return slice
end proc;
proc Array_sort(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
if phase = compile then
    throw a ConstantError exception — sort cannot be called from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.
if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
end if;

Evaluate ??? and ignore its result
end proc;

proc Array_splice(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Object
if phase = compile then
    throw a ConstantError exception — splice cannot be called from a constant expression
end if;

note This function is generic and can be applied even if this is not an Array.
if |args| < 2 then
    throw an ArgumentError exception — at least two arguments must be supplied
end if;

length: INTEGER [] readLength(this, phase);
startArg: Object [] defaultArg(args, 0, +zero_64);
deleteCountArg: Object [] defaultArg(args, 1, +zero_64);
start: INTEGER [] pinExtendedInteger(objectToExtendedInteger(startArg, phase), length, true);
deleteCount: INTEGER [] pinExtendedInteger(objectToExtendedInteger(deleteCountArg, phase), length – start, false);
deletedSlice: Object [] makeArraySlice(this, start, start + deleteCount, phase);
newElts: Object [] args[2 ...];
newEltsCount: INTEGER [] newElts.length;
countDiff: INTEGER [] newEltsCount – deleteCount;
i: INTEGER;
if countDiff < 0 then
    i = start + deleteCount;
    while i ≤ length do
        elt: ObjectOpt [] indexRead(this, i, phase);
        Evaluate indexWrite(this, i + countDiff, elt, phase) and ignore its result;
        i = i + 1
    end while;
    i = 0;
    while i ≤ countDiff do
        i = i – 1;
        Evaluate indexWrite(this, length + i, none, phase) and ignore its result
    end while
elsif countDiff > 0 then
    i = length;
    while i ≤ start + deleteCount do
        i = i – 1;
        elt: ObjectOpt [] indexRead(this, i, phase);
        Evaluate indexWrite(this, i + countDiff, elt, phase) and ignore its result
    end while
end if;

Evaluate writeLength(this, length + countDiff, phase) and ignore its result;

for each arg [] newElts do
    Evaluate indexWrite(this, i, arg, phase) and ignore its result;
    i = i + 1
end for each;
return deletedSlice
end proc;
proc $Array\_unshift(this: Object, f: SimpleInstance, args: Object[], phase: Phase): Float64$

if $phase = \text{compile}$ then
    throw a $ConstantError$ exception — $unshift$ cannot be called from a constant expression
end if;

\textit{note} This function is generic and can be applied even if $this$ is not an $Array$.

i: Integer $\leftarrow$ readLength($this$, $phase$);
nArgs: Integer $\leftarrow$ $\vert$ args $\vert$;
newLength: Integer $\leftarrow$ $nArgs + i$;

if $nArgs = 0$ then
    At the implementation’s discretion, either do nothing or return $newLength_{64}$
end if;

while $i \neq 0$ do
    $i \leftarrow i - 1$;
    elt: ObjectOpt $\leftarrow$ indexRead($this$, $i$, $phase$);
    Evaluate $indexWrite(this, i + nArgs, elt, phase)$ and ignore its result
end while;

for each $arg \in$ args do
    Evaluate $indexWrite(this, i, arg, phase)$ and ignore its result;
    $i \leftarrow i + 1$
end for each;

return $newLength_{64}$
end proc;

17.14 Namespace

$Namespace$: Class = new Class{{localBindings: {}}, instanceProperties: {}}, super: Object,
prototype: NamespacePrototype, complete: true, name: “Namespace”, typeofString: “namespace”,
dynamic: false, final: true, defaultValue: null, defaultHint: hintString, hasProperty: ordinaryHasProperty,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: ordinaryCall, construct: constructNamespace, init: none, is: ordinaryIs,
coerce: ordinaryCoerce
proc constructNamespace(c: CLASS, args: OBJECT[], phase: PHASE): NAMESPACE

note This function can be used in a constant expression if its argument is a string.
if \(|args| > 1\) then
    throw an ArgumentError exception — at most one argument can be supplied
end if;
arg: OBJECT[] defaultArg(args, 0, undefined);
if arg \[\] NULL \[\] UNDEFINED then
    if phase = compile then
        throw a ConstantError exception — a constant expression cannot construct new anonymous namespaces
    end if;
    return new NAMESPACE[name: “anonymous”]
elsearg \[\] CHAR16 \[\] STRING then
    name: STRING[] toString(arg);
    if name = “” then return public
    elsif some ns \[\] namedNamespaces satisfies ns.name = name then return ns
    else
        ns2: NAMESPACE[] new NAMESPACE[name: name]
        namedNamespaces \[\] namedNamespaces \[\] {ns2};
        return ns2
    end if
else throw a TypeError exception
end if
end proc;

namedNamespaces: NAMESPACE\[\] \{\};

NamespacePrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
    stdFunction(public::“toString”, Namespace_toString, 0),
    stdReserve(public::“valueOf”, ObjectPrototype),
    archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
    env: none\]\];

proc Namespace_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING

note This function does not check phase and therefore can be used in a constant expression.
note This function ignores any arguments passed to it in args.
if this \[\] NAMESPACE then throw a TypeError exception end if;
return this.name
end proc;

17.15 Attribute

Attribute: CLASS = new CLASS[localBindings: {}, instanceProperties: {}, super: Object, prototype: ObjectPrototype,
    complete: true, name: “Attribute”, typeofString: “object”, dynamic: false, final: true,
    defaultValue: null, defaultHint: hintText, hasProperty: ordinaryHasProperty,
    bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
    bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
    enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: none, is: ordinaryIs,
    coerce: ordinaryCoerce]
17.16 Date


17.17 RegExp


17.18 Class


classPrototypeGetter: **INSTANCEGETTER** = new **INSTANCEGETTER**[fullname: {public:"prototype"}], final: true, enumerable: false, call: **Class_prototype**

proc **Class_prototype**(this: **Object**, phase: **PHASE**): **OBJECT**

    note **this** [] **CLASS** because this getter cannot be extracted from the **CLASS** class.

    proto **OBJECTOPT** [] **this.prototype**;

    if **prototype** = none then return undefined else return **prototype** end if
end proc;

**ClassPrototype**: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**[localBindings: {}

    stdConstBinding(public:"constructor", **Class**, **Class**),
    stdFunction(public:"toString", **Class_toString**, 0),
    stdReserve(public:"valueOf", **ObjectPrototype**),
    stdConstBinding(public:"length", **Number**, [164]),
proc Class_toString(this: OBJECT, f: SIMPLE_INSTANCE, args: OBJECT[], phase: PHASE): STRING
note This function does not check phase and therefore can be used in a constant expression.
note This function ignores any arguments passed to it in args.
c: CLASS[] = objectToClass(this);
return "[class " ⊕ c.name ⊕ "]"
end proc;

17.19 Function

Function: CLASS = new CLASS[]localBindings: {}, instanceProperties: {ivarFunctionLength}, super: Object,
prototype: FunctionPrototype, complete: true, name: "Function", typeofString: "function",
dynamic: false, final: true, defaultValue: null, defaultHint: hintString, hasProperty: ordinaryHasProperty,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete,
read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: none, is: ordinaryIs,
coerce: ordinaryCoerce[]

ivarFunctionLength: INSTANCE_VARIABLE = new INSTANCE_VARIABLE[]multiname: {public::"length"}, final: true,
enumerable: false, type: Number, defaultValue: none, immutable: true[]

FunctionPrototype: SIMPLE_INSTANCE = new SIMPLE_INSTANCE[]localBindings: {}, archetype: ObjectPrototype,

17.19.1 PrototypeFunction

PrototypeFunction: CLASS = new CLASS[]localBindings: {}, instanceProperties: 
{new INSTANCE_VARIABLE[]multiname: {public::"prototype"}, final: true, enumerable: false, type: Object,
defaultValue: undefined, immutable: false[]}, super: Function, prototype: FunctionPrototype, complete: true,
name: "Function", typeofString: "function", dynamic: true, final: true, defaultValue: null,
defaultHint: hintString, hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead,
bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall,
construct: dummyConstruct, init: none, is: ordinaryIs, coerce: ordinaryCoerce[]

17.20 Package

Package: CLASS = new CLASS[]localBindings: {}, instanceProperties: {}, super: Object, prototype: ObjectPrototype,
complete: true, name: "Package", typeofString: "object", dynamic: true, final: true, defaultValue: null,
defaultHint: hintString, hasProperty: ordinaryHasProperty, bracketRead: ordinaryBracketRead,
bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall,
construct: dummyConstruct, init: none, is: ordinaryIs, coerce: ordinaryCoerce[]
17.21 Error

```
Error: CLASS = new CLASS[localBindings: {}, instanceProperties: {
}
}

proc callError(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
  if [args] > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT[] defaultArg(args, 0, undefined);
  if arg = null or is(arg, Error) then return arg
  else return construct(c, args, phase)
  end if
end proc;

proc initError(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
  if [args] > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  name: STRING[] NULL[] dotWrite(ErrorPrototype, {public: “name”}, phase);
Evaluate dotWrite(this, {public: “name”}, name, phase) and ignore its result;
  arg: OBJECT[] defaultArg(args, 0, undefined);
  message: STRING[] NULL;
  if arg = undefined then
    message[] dotWrite(ErrorPrototype, {public: “message”}, phase)
  else message[] objectToString(arg, phase)
  end if;
Evaluate dotWrite(this, {public: “message”}, message, phase) and ignore its result
end proc;

ErrorPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE[localBindings: {
  stdConstBinding(public: “constructor”, Class, Error),
  stdFunction(public: “toString”, Error_toString, 1),
  stdVarBinding(public: “name”, String, “Error”),
  stdVarBinding(public: “message”, String, an implementation-defined string)),
}

proc Error_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): STRING
  if phase = compile then
    throw a ConstantError exception — toString cannot be called on an Error from a constant expression
  end if;
  note This function ignores any arguments passed to it in args.
  err: OBJECT[] coerceNonNull(this, Error);
  name: STRING[] NULL[] dotRead(err, {public: “name”}, phase);
  message: STRING[] NULL[] dotRead(err, {public: “message”}, phase);
  return an implementation-defined string derived from name, message, and optionally other properties of err
end proc;
```
### 17.21.1 Error Subclasses

```plaintext
proc makeBuiltInErrorSubclass(name: STRING): CLASS
proc call(this: OBJECT, c: CLASS, args: OBJECT[], phase: PHASE): OBJECT
if |arg| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
end if;
arg: OBJECT [] defaltArg(args, 0, undefined);
if arg = null or is(arg, Error) then return coerce(arg, c)
else return construct(c, args, phase)
end if
end proc;
c: CLASS [] new CLASS[localBindings: {}, instanceProperties: {}, super: Error, complete: false, name: name,
typeOfString: "object", dynamic: true, final: false, defaultValue: null, defaultHint: hintNumber,
hasProperty: Error.hasOwnProperty, bracketRead: Error.bracketRead, bracketWrite: Error.bracketWrite,
bracketDelete: Error.bracketDelete, read: Error.read, write: Error.write, delete: Error.delete,
enumerate: Error.enumerate, call: call, construct: ordinaryConstruct, init: none, is: ordinaryIs,
coerce: ordinaryCoerce[]
prototype: SIMPLEINSTANCE [] new SIMPLEINSTANCE[localBindings: {}
stdConstBinding(public::"constructor", Class, c),
stdVarBinding(public::":name", String, name),
stdVarBinding(public::":message", String, an implementation-defined string)
archetype: ErrorPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
env: none[]
proc init(this: SIMPLEINSTANCE, args: OBJECT[], phase: {run})
if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
end if;
name2: STRING [] NULL [] dotRead(prototype, {public::":name"}, phase);
Evaluate dotWrite(this, {public::":name"}, name2, phase) and ignore its result;
arg: OBJECT [] defaultArg(args, 0, undefined);
message: STRING [] NULL;
if arg = undefined then message [] dotRead(prototype, {public::":message"}, phase)
else message [] objectToString(arg, phase)
end if;
Evaluate dotWrite(this, {public::":message"}, message, phase) and ignore its result
end proc;
c.prototype [] prototype;
c.init [] init;
c.complete [] true;
return c
end proc;

ArgumentError: CLASS = makeBuiltInErrorSubclass("ArgumentError");
AttributeError: CLASS = makeBuiltInErrorSubclass("AttributeError");
ConstantError: CLASS = makeBuiltInErrorSubclass("ConstantError");
DefinitionError: CLASS = makeBuiltInErrorSubclass("DefinitionError");
EvalError: CLASS = makeBuiltInErrorSubclass("EvalError");
RangeError: CLASS = makeBuiltInErrorSubclass("RangeError");
```
ReferenceError: CLASS = makeBuiltInErrorSubclass("ReferenceError");

SyntaxError: CLASS = makeBuiltInErrorSubclass("SyntaxError");

TypeError: CLASS = makeBuiltInErrorSubclass("TypeError");

UninitializedError: CLASS = makeBuiltInErrorSubclass("UninitializedError");

URIError: CLASS = makeBuiltInErrorSubclass("URIError");

## A Index

### A.1 Nonterminals

- AdditiveExpression 113
- AnnotatableDirective 152
- Arguments 106
- ArrayLiteral 100
- ASCIIDigit 34
- AssignmentExpression 124
- Attribute 155
- AttributeCombination 155
- AttributeExpression 102
- Attributes 155
- BitwiseAndExpression 120
- BitwiseOrExpression 120
- BitwiseXorExpression 120
- Block 133
- BlockCommentCharacters 30
- Brackets 106
- BreakStatement 147
- CaseElement 136
- CaseElements 136
- CaseElementsPrefix 136
- CaseLabel 136
- CatchClause 149
- CatchClauses 149
- CatchClausesOpt 149
- ClassDefinition 182
- CompoundAssignment 124
- ConditionalExpression 123
- ContinueStatement 146
- ContinuingIdentifierCharacter 31
- ContinuingIdentifierCharacterOrEscape 31
- ControlEscape 36
- DecimalDigits 34
- DecimalIntegerLiteral 34
- DecimalLiteral 33
- Directive 152
- Directives 152
- DirectivesPrefix 152
- DivisionPunctuator 33

- DoStatement 139
- ElementList 100
- EmptyStatement 132
- EndOfFile 28
- EqualityExpression 118
- ExpressionQualifiedIdentifier 92
- ExpressionStatement 132
- ExpressionsWithRest 106
- FieldList 98
- FieldName 98
- ForInBinding 141
- ForInitializer 141
- ForStatement 141
- Fraction 34
- FullNewExpression 102
- FullNewSubexpression 103
- FullPostfixExpression 102
- FunctionCommon 166
- FunctionDefinition 166
- FunctionExpression 97
- FunctionName 166
- HexDigit 34
- HexEscape 36
- HexIntegerLiteral 34
- Identifier 92
- IdentifierName 31
- IdentifierOrKeyword 30
- IdentityEscape 36
- IfStatement 135
- ImportDirective 157
- Inheritance 182
- InitialIdentifierCharacter 31
- InitialIdentifierCharacterOrEscape 31
- InputElement 28
- IntegerLiteral 33
- LabeledStatement 134
- LetterE 33
- LetterF 33

- LetterL 33
- LetterU 33
- LetterX 34
- LineBreak 29
- LineBreaks 29
- LineComment 30
- LineCommentCharacters 30
- LineTerminator 29
- ListExpression 127
- LiteralElement 100
- LiteralField 98
- LiteralStringChar 36
- LogicalAndExpression 122
- LogicalAssignment 125
- LogicalOrExpression 122
- LogicalXorExpression 122
- Mantissa 33
- MultiLineBlockComment 30
- MultiLineBlockCommentCharacters 30
- MultiplicativeExpression 111
- NamespaceDefinition 184
- NextInputElement 28
- NonAssignmentExpression 123
- NonemptyFieldList 98
- NonemptyParameters 177
- NonTerminator 30
- NonTerminatorOrAsteriskOrSlash 30
- NonTerminatorOrSlash 30
- NonZeroDecimalDigits 34
- NonZeroDigit 34
- NullEscape 31
- NullEscapes 31
- NumericLiteral 33
- ObjectLiteral 98
- OptionalExpression 141
- OrdinaryRegExpChar 37
- PackageDefinition 185
- PackageIdentifiers 185
A.2 Tags

→ 12, 13
++ 12, 13
+zero 12, 13
andEq 125
busy 50
compile 46
constructorFunction 47
default 46
equal 52
false 4, 39
final 40
forbidden 49, 50
get 47
greater 52
hintNumber 52
hintString 52
instanceFunction 47
less 52
MinusInfinity 52
MinusZero 52
NaN 12, 13, 52
none 39, 40, 42, 47, 48, 50, 51
normal 47
null 39
orEq 125
plainFunction 47
PlusInfinity 52
PlusZero 52
prototypeFunction 47
read 49
readWrite 49
run 46
set 47
static 40
true 4, 39
uncheckedFunction 47
undefined 39
unordered 52
virtual 40
write 49
xorEq 125
–zero 12, 13

A.3 Semantic Domains
A.4 Globals

accessesOverlap 66
add 113
ancestors 58
archetype 66
archetypes 66
ArgumentError 225
Array 212
Array_concat 215
Array_getLength 213
Array_join 216
Array_pop 216
Array_push 217
Array_reverse 217
Array_setLength 214
Array_shift 218
Array_slice 218
Array_sort 219
Array_splice 219
Array_toLocaleString 215
Array_toString 214
Array_unshift 220
arrayLimit 212
arrayPrivate 212
ArrayPrototype 214
assignArguments 176
Attribute 221
AttributeValue 225
bitAnd 121
bitNot 111
bitOr 122
bitwiseAnd 7
bitwiseOr 7

bitwiseShift 7
bitwiseXor 7
bitXor 121
Boolean 196
Boolean_toString 197
BooleanPrototype 197
byte 204
call 83
callChar 205
callError 224
callInit 84
callNull 196
callObject 193
callVoid 196
cannotReturnValue 148
classChar 205
classIntegerCode 205
class16ToInteger 8
class21ToInteger 8
class21ToUTF16 56
classPrototype 206
classToLowerFull 57
classToLowerCase 57
classToUpperCase 57
classToUpperCase 57
classToUpperCase 57
classToLowerFull 57
classToLowerCase 57
classToLowerCase 57
classToUpperCase 57
checkAccessorParameters 175
checkInteger 54
Class 222
Class_prototype 222
Class_toString 223
ClassPrototype 222
classPrototypeGetter 222

coop 64
coopBoolean 197
coopChar 205
coopFloat 202
coopGeneralNumber 197
coopLong 201
coopNever 195
coopNonNull 65
coopNull 196
coopNumber 203
coopObject 193
coopOr 64
coopString 207
coopULong 202
coopVoid 196
combineAttributes 65
ConstantError 225
construct 84
constructBoolean 197
constructChar 205
constructFloat 202
constructGeneralNumber 197
constructLong 201
constructNamespace 221
constructNever 195
constructNull 196
constructNumber 203
constructObject 193
constructString 206
constructULong 201
constructVoid 196
createDynamicProperty 79
RangeError 225  
rationalToLong 54  
rationalToULong 54  
readArrayPrivateLength 212  
readImplicitThis 70  
readInstanceProperty 74  
readInstanceSlot 73  
readLength 71  
readReference 71  
readSingletonProperty 75  
readString 206  
realToFloat32 12  
realToFloat64 14  
ReferenceError 226  
RegExp 222  
RegExpPrototype 222  
remainder 112  
sameAsConstruct 83  
sbyte 204  
sealAllLocalProperties 90  
sealLocalProperty 90  
sealObject 90  
searchForOverrides 86  
ssetupVariable 66  
shiftLeft 115  
shiftRight 115  
shiftRightUnsigned 116  
short 204  
ssignatureLength 177  
signedWrap32 53  
signedWrap64 53  
stdConstBinding 91  
stdExplicitConstBinding 91  
stdFunction 91  
stdVarBinding 91  
String 206  
String_charAt 208  
String_charCodeAt 208  
String_concat 208  
String_fromCharCode 207  
String_indexOf 208  
String_lastIndexOf 209  
String_length 207  
String_locateCompare 209  
String_match 209  
String_replace 210  
String_search 210  
String_slice 210  
String_split 210  
String_substring 211  
String_toLocaleLowerCase 211  
String_toLocaleUpperCase 212  
String_toLowerCase 211  
String_toString 207  
String_toUpperCase 211  
stringHasProperty 206  
stringPrefixToInteger 191  
StringPrototype 207  
stringToExtendedInteger 61  
stringToFloat32 61  
stringToFloat64 61  
stringToUTF32 56  
subtract 114  
surrogatePairToSupplementaryChar 56  
SyntaxError 226  
systemError 225  
toCompoundAttribute 65  
toFloat32 55  
toFloat64 55  
toRational 55  
toString 62  
truncateFiniteFloat32 13  
truncateFiniteFloat64 14  
truncateToInteger 53  
TypeError 226  
uint 204  
ulong 201  
ulongPrototype 202  
UninitializedError 226  
unsignedWrap32 52  
unsignedWrap64 53  
URIError 226  
ushort 204  
Void 195  
writeArray 212  
writeArrayPrivateLength 213  
writeInstanceProperty 80  
writeLength 76  
writeReference 76  
writeSingletonProperty 80  
writeVariable 67